Aktoripohjainen pelimoottoriarkkitehtuuri

Antti Hietasaari

Helsinki 20.04.2016

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos
Pro gradu -tutkielma

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet –Faculty	Laitos – Institution – Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author

Antti Hietasaari

Työn nimi – Arbetets titel – Title

Aktoripohjainen pelimoottoriarkkitehtuuri

Oppiaine – Läroämne – Subject

Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu -tutkielma	20.04.2016	62

Tiivistelmä – Referat – Abstract

Tutkielmassa arvioidaan aktoripohjaisten rinnakkaistamisratkaisujen soveltuvuutta pelimoottoreihin. Tutkielmassa esitellään ensin pelimoottoreiden ja aktoripohjaisen rinnakkaisuuden perusperiaatteet ja sitten aktoripohjainen Stage-pelimoottoritoteutus. Tutkielman lopuksi tutkitaan Stage-moottorin tehokkuutta ja helppokäyttöisyyttä verrattuna perinteisiä lukkopohjaisia rinnakkaistamisratkaisuja hyödyntävään pelimoottoriin.

ACM Computing Classification System (CCS):
• **Applied computing~Computer games**
• *Computing methodologies~Concurrent computing methodologies*
• *Software and its engineering~Software architectures*

Avainsanat – Nyckelord – Keywords

pelimoottorit, videopelit, rinnakkaisuus, aktorit, ohjelmistoarkkitehtuuri

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto

Videopelit ovat perinteisesti olleet ohjelmistokehityksen osa-alue, jolla tehokkuudella on suuri merkitys. Pelit ovat jatkuvasti pelaajan kanssa vuorovaikutuksessa olevia simulaatioita, joissa pienelläkin hidastumisella saattaa olla suuri vaikutus pelielämykseen ja joissa kilpailijoita hienompi grafiikka on usein ollut tärkeä myyntivaltti, joten pelialalla on sen alkuajoista saakka ollut tärkeää saada laitteistosta irti mahdollisimman paljon. Tämän vuoksi pelialalla on ryhdytty hyödyntämään rinnakkaisohjelmointia monia muita ohjelmistokehityksen aloja innokkaammin: koska laitteiston suoritustehoa kasvatetaan nykyään lähinnä lisäämällä rinnakkaisten suorittimien määrää, vaaditaan sen tehokkaaseen hyödyntämiseen myös ohjelmiston rinnakkaistamista.

Rinnakkaisohjelmointi alkoi pelialalla yleistyä toden teolla seitsemännen konsolisukupolven alussa (suurin piirtein vuosina 2005-2007), koska sekä Sonyn PlayStation 3 että Microsoftin Xbox 360 -konsoleissa käytettiin moniydinsuorittimia. Samoihin aikoihin niin kutsuttujen pelimoottoreiden käyttö lisääntyi alalla selvästi. Pelimoottori on pelien kehittämiseen tarkoitettu ohjelmistokehikko: se tarjoaa palveluita, joita tarvitaan monenlaisissa eri peleissä, kuten esimerkiksi grafiikan piirtoa, fysiikkamallinnusta tai vaikkapa Internet-rajapintaa moninpeliä varten. Kun tärkeimmät palvelut on toteutettu pelimoottorissa, niitä ei tarvitse ohjelmoida uudestaan jokaista peliä varten, vaan uuden pelin luomiseen riittää pelisääntöjen ja -sisällön toteuttaminen valmiin kehyksen päälle. Pelimoottoreita on käytetty jo alalla jo pitkään, mutta toisaalta vielä runsas vuosikymmen sitten oli yleistä, että peli toteutettiin puhtaalta pöydältä tai muokkaamalla aikaisempaa peliä. Vasta seitsemännen konsolisukupolven tuomat korkeammat kehityskustannukset pakottivat valtaosan kehittäjistä käyttämään valmiita moottoreita – nykyisin valtaosa miljoonia kappaleita myyvistä ja paljon mediahuomiota saavista ns. AAA-peleistä on rakennettu jonkin valmiin pelimoottorin, kuten Epic Gamesin Unreal Enginen tai Crytekin CryEnginen, päälle.

Koska pelimoottoreita käytetään monissa eri peleissä, on niiden tehokas toteutus erittäin tärkeää – nykyään tämä tarkoittaa, että moottorissa on hyödynnettävä rinnakkaisuutta. Useimmissa pelimoottoreissa rinnakkaistaminen on toteutettu perinteisten rinnakkaisuudenhallintamekanismien kuten semaforien ja luku/kirjoituslukkojen avulla. Tässä tutkielmassa tutkitaan, miten eräs moderneista rinnakkaisuudenhallintamekanismeista,

aktorimalli, soveltuu pelimoottorien toteuttamiseen. Luvussa 2 esitellään tarkemmin pelimoottorin käsite ja historia sekä tarkastellaan perinteisiä pelimoottorien rinnakkaistamistekniikoita. Luku 3 käy läpi joitakin rinnakkaistamisen pelimoottoritoteutuksissa aiheuttamia ongelmia, ja luvussa 4 käydään läpi aktorimallin käsite ja toimintaperiaate. Luku 5 esittelee aktoripohjaisen Stage-pelimoottorin ja käy läpi sen toteutuksen haasteita ja ratkaisuja, ja luvussa 6 arvioidaan aktoripohjaisen pelimoottoritoteutuksen toimivuutta perinteisiin lukkopohjaisiin rinnakkaistamisratkaisuihin verrattuna.

2 Pelimoottorit

Tässä luvussa käsitellään lyhyesti pelimoottorien ideaa, historiaa ja yleistä arkkitehtuuria.

2.1 Periaate ja historia

Pelimoottori on sovelluskehys, jonka avulla voidaan toteuttaa useita eri pelejä ilman, että kaikki perusominaisuudet on ohjelmoitava erikseen jokaista peliä varten. Pelimoottorien toteutuksessa painotetaan usein modulaarisuutta – eri palvelut on yleensä toteutettu omina moduuleinaan, joita voidaan ottaa käyttöön, muokata, korvata tai poistaa jokaisen pelin yksilöllisten vaatimusten mukaan [GrJ09, s.11-13]. Esimerkiksi vuoropohjaisessa strategiapelissä ei välttämättä ole lainkaan tarvetta fysiikkamoottorille ja tasoloikkapelin toteuttaja saattaa haluta toteuttaa oman fysiikkamallinnusmoduulinsa, jotta pelin kontrolleihin saadaan juuri kehittäjän haluama tuntu.

Pelimoottorin ytimessä on niin kutsuttu pelisilmukka [TBN06]. Silmukan jokaisella suorituskerralla kerätään käyttäjän syötteet, lasketaan pelisimulaation uusi tila syötteiden, tekoälyn, fysiikkamoottorin ja muiden pelin sääntöjen perusteella ja lopuksi piirretään simulaation tulos ruudulle. Kun silmukkaa suoritetaan tarpeeksi usein (yleensä joko 30 tai 60 kertaa sekunnissa), muodostuu illuusio reaaliajassa toimivasta interaktiivisesta pelimaailmasta.

Pelialan alkuaikoina moottoreita ei juurikaan käytetty, vaan oli yleistä toteuttaa jokainen peli puhtaalta pöydältä. Koska pelit ja laitteistot olivat yksinkertaisia, ei toiminnallisuuden toteuttaminen uudelleen ollut valtavan kallista, ja alustojen ollessa tehottomia oli järkevää tehdä paljon yksittäisten pelien tarpeisiin kohdennettua optimointia.

Ensimmäinen pelimoottorien suuri läpilyönti oli Id Softwaren *Doom* vuodelta 1993. Doomissa käytettiin aikanaan erikoista datajohtoista (data-driven) arkkitehtuuria: pelin ydintoiminnallisuus kuten grafiikan piirto, äänet ja törmäyksentunnistus oli erotettu varsinaisesta pelisisällöstä [GrJ09, s.11]. Muokkaamalla pelin kentät ja resurssit sisältävää WAD-tiedostoa Doomista saattoi tehdä täysin eri pelin ilman, että muokkaajan tarvitsi koskea vaikeatajuisempien moduulien toteutukseen tai edes ymmärtää niiden toimintaa syvällisesti. Pelin ympärille muodostuikin suuri yhteisö, jossa käyttäjät jakoivat niin

kutsuttuja modeja eli itse tekemiään muokkauksia, jotka lisäsivät peliin uutta sisältöä, kuten aseita tai kenttiä, tai jopa muuttivat sen kokonaan uudeksi peliksi. Moottoria myös lisensoitiin muille ammattilaiskehittäjille, ja sen päälle rakennettiin muun muassa Raven Softwaren *Heretic* (1994) ja *Hexen* (1995).

Doomin ajoista eteenpäin saatavilla olevien pelimoottorien määrä ja käyttö on kasvanut tasaisesti, mutta moottorien todellinen läpilyönti tapahtui edellisen, seitsemännen konsolisukupolven aikana. Nykypelien budjettien noustessa korkeimmillaan jopa satoihin miljooniin euroihin [ReD14] koodin kierrättämisestä saatavat säästöt ovat tulleet tarpeeseen, eikä kaikilla kehitysstudioilla enää välttämättä edes riitä ammattitaitoa kaikkein näyttävimpien visuaalisten efektien tuottamiseen tai monimutkaisimpien nykypeleissä hyödynnettävien algoritmien toteuttamiseen. Samalla kasvaneet tehot ovat mahdollistaneet monikäyttöisemmät moottorit: siinä missä Doomin pohjalta kyettiin kehittämään vain alkuperäisen pelin kaltaisia pelihahmon näkökulmasta kuvattuja räiskintöjä, nykyiset kaupalliset moottorit kuten Unreal Engine, Unity tai Cryengine taipuvat lähes mihin tahansa lajityyppiin. Näiden seikkojen seurauksena valtaosa pelinkehittäjistä on jo siirtynyt käyttämään lisensoituja pelimoottoreita [DeM09].

2.2 Pelimoottorimoduulit

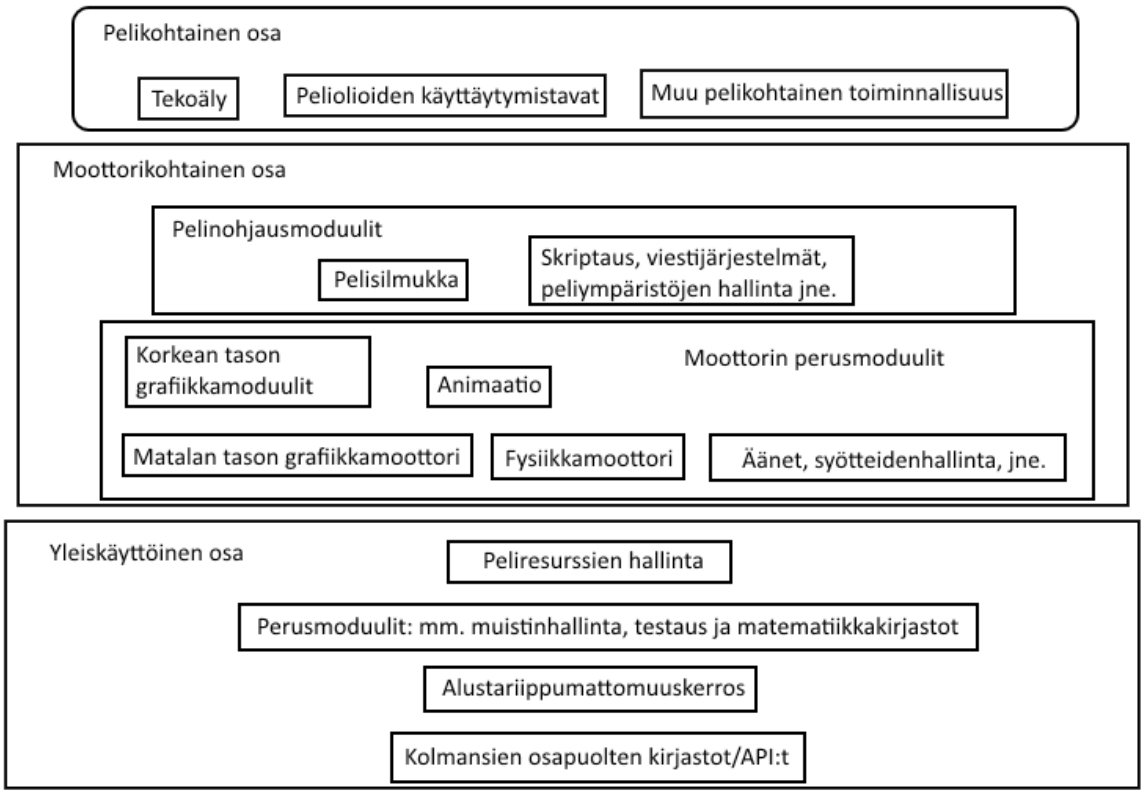
Pelimoottorin toteutukseen on monia eri lähestymistapoja, mutta yleiskäyttöisille moottoreille ovat yhteisiä modulaariset komponentit. Jos koko nykypelien kirjon halutaan olla toteutettavissa, kehittäjillä on oltava mahdollisuus muokata lähes mitä tahansa moottorin toiminnallisuutta, minkä vuoksi eri toimintoja toteuttavien moottorin osien on oltava helposti vaihdettavissa. Tässä tutkielmassa näitä komponentteja kutsutaan useimmista muista lähteistä poiketen pääsääntöisesti moduuleiksi, jotta vältytään sekaannuksilta pelioliokomponenttien (ks. luku 2.3, Peliolioiden koostaminen) kanssa.

Moottorin moduulit muodostavat usein kerrosarkkitehtuurin, jonka jokainen kerros tarjoaa rajapinnat, joiden kautta varsinaisia komponentteja käytetään [GrJ09, s.28-29]. Alimmilla tasoilla olevat moduulit tarjoavat yleensä rajapintoja laitteistoon ja muuttamalla niiden toteutusta pelimoottori voidaan siirtää uusille alustoille. Korkeimmilla tasoilla ovat puolestaan pelikohtaiset moduulit, joiden avulla on toteutettu yksittäisten pelien säännöt ja jotka vaihtamalla moottorin päälle voidaan rakentaa uusia pelejä. Väliin jäävät tasot ovat pelimoottorin varsinainen ydin: ne tarjoavat yleishyödyllisiä,

pelistä ja laitteistosta riippumattomia palveluita ja esiintyvät usein muuttumattomina valtaosassa moottorin avulla tuotetuista ohjelmista. Yksinkertaistettu kaavio moottorin moduuleista on nähtävissä kuvassa 2.1.

Moduulikerrokset

Moottorin pohjana toimivat erilaiset kolmansien osapuolien kirjastot ja ohjelmointirajapinnat kuten OpenGL, Direct X, Havok tai vaikkapa Boost, jotka tarjoavat matalan tason laitteistoläheisiä palveluita esimerkiksi grafiikan tai fysiikan laskemiseen. Koska kaikkia kirjastoja ei ole saatavilla kaikilla alustoilla (esimerkiksi Direct X toimii vain Microsoftin laitteissa ja käyttöjärjestelmissä), moottori tarvitsee siirrettävyyden vuoksi yleensä alustariippumattomuuskerroksen, joka tunnistaa käytössä olevat kirjastot, valitsee niistä tarvittavat ja toimii tulkkina niiden ja korkeamman tason moduulien välillä [GrJ09, s.34]. Tämän päällä ovat varsinaisen moottorin perusmoduulit: kirjastot, jotka tarjoavat palveluita esimerkiksi muistinhallintaan, testaukseen, moottorin osien ajoaikaiseen lataamiseen ja moottorin tarvitsemien matemaattisten operaatioiden laskemiseen [GrJ09, s.34-35]. Näiden päälle puolestaan rakentuu moottorin resurssienhallinta, joka on vastuussa dataresurssien kuten tekstuurien tai komentosarjojen (skriptien) lataamisesta, vapauttamisesta ja hallinnoinnista [GrJ09, s.35-36].



Kuva 2.1: Pelimoottorin ja pelin moduulikerrokset [yksinkertaistettu lähteestä GrJ09, s.29]

Edellä mainitut moduulit ovat käyttötarkoitukseltaan melko yleisiä eivätkä välttämättä erityisesti liity juuri peleihin. Ne toimivat kuitenkin pohjana seuraavalle moottorin kerrokselle, joka sisältää peleille hyvin keskeisiä moduuleja kuten törmäysten ja fysiikan laskennan, äänimoottorin, syötteidenhallinnan ja matalan tason grafiikkamoottorin [GrJ09, s.36-45]. Fysiikka- ja grafiikkamoottori tarjoavat perustukset moottorin animaatiomoduulille, ja koska matalan tason grafiikkamoottori mahdollistaa yleensä vain hyvin yksinkertaisia piirtopalveluita, se tarvitsee päälleen muita moduuleja esimerkiksi polygonien karsintaan sekä sävyttimien ja muiden monimutkaisten visuaalisten efektien tuottamiseen [GrJ09, s.37-43]. Kaiken tämän päällä toimivat pelinohjausmoduulit, jotka tarjoavat muun muassa mahdolliset skriptaus- ja viestijärjestelmät, peliympäristöjen lataus- ja vaihtopalvelut sekä itse varsinaisen pelisilmukan [GrJ09, s.45-48].

Muiden moduulien yläpuolella ovat pelikohtaiset moduulit kuten tekoäly sekä pelaajan hahmon liikkeitä ja pelimekaniikkoja hallinnoivat järjestelmät [GrJ09, s.48-49]. Yleensä nämä eivät kuulu varsinaiseen moottoriin vaan ne jätetään pelinkehittäjien huoleksi, sillä niiden toiminta on useimmiten liian pelikohtaista sopiakseen osaksi yleiskäyttöistä moottoria. Myös pelikohtaisten moduulien moottorimainen uusiokäyttö on tietysti yleistä esimerkiksi pelien jatko-osissa, joissa pelimekaniikat muistuttavat paljon alkuperäistä peliä.

2.3 Peliolioiden koostaminen

Perinteisesti olio-ohjelmoinnissa eri tietotyyppien jakamaa yhteistä toiminnallisuutta on mallinnettu perinnän avulla: jos kaksi luokkaa käyttäytyy osittain samalla tavalla, toinen asetetaan perimään toisen tai jaettu toiminnallisuus mallinnetaan erillisellä luokalla, jonka molemmat perivät. Peleissä tällainen järjestely ei kuitenkaan usein ole järkevä, sillä pelioliot asettuvat harvoin siististi perintäpuuhun [BiS02]. Esimerkiksi useimmissa reaaliaikastrategiapeleissä pelaajalla on komennettavissaan sekä rakennuksia että ns. yksiköitä, kuten esimerkiksi jalkaväkeä ja tankkeja. Näillä on joitakin yhteisiä ominaisuuksia, kuten esimerkiksi kestävyys, mutta molemmilla on myös niille ominaisia toimintoja: pelaaja voi liikuttaa yksiköitä kartalla ja rakennukset voivat tuottaa uusia yksiköitä. Selvästikin sekä yksiköt että rakennukset ovat erikoistuksia samasta yliluokasta, mutta miten käy, jos kehittäjä päättää luoda liikkuvan tehdasyksikön, joka kykenee sekä liikkumaan kartalla että luomaan muita yksiköitä? Jotta molemmat ominaisuudet saatai-

siin samalle oliolle, on joko käytettävä moniperintää, mikä ei ole mahdollista monissa kielissä, tai sovitettava rakennukset ja yksiköt samaan perintäketjuun. Tällöin pelioliot saavat väistämättä turhaa dataa ja ominaisuuksia, joita ei koskaan käytetä: joko kaikille rakennuksille on annettava liikkumisen mahdollistavat metodit tai kaikille yksiköille yksiköntuotantotoiminnallisuus. Ratkaisuna monissa pelimoottoreissa oliot muodostetaankin perinnän sijasta koostamalla [PaE09].

Koostaminen toimii sisällyttämällä varsinaiseen peliolioon vain keskeisimmät, kaikille peliolioille yhteiset toiminnot ja toteuttamalla kaikki muut erityisten pelioliokomponenttien sisällä. Jokaisella oliolla on lista, tietokantarivi tai muu tietorakenne, joka sisältää kaikki kyseisen olion käyttämät komponentit [BiS02]. Kun olio päivittää tilaansa, se käy listan läpi ja kutsuu kaikkien komponenttiensa päivitysmetodeja. Muut oliot voivat käyttää komponenttien tarjoamia palveluita pyytämällä oliolta viitettä haluttuun komponenttiin. Näin jokaiselle oliolle saadaan juuri ne toiminnot, joita se tosiasiassa myös käyttää. Menetelmä mahdollistaa myös toiminnallisuuden lisäämisen ja poistamisen suoritusaikana lisäämällä tai poistamalla komponentteja olion komponenttilistasta. Lisäksi tehokkuudelle vähemmän kriittiset käyttäytymistavat voidaan liittää olioihin skriptikielillä toteutettuina komponentteina, jolloin prototyyppejä voidaan iteroida nopeasti ilman, että osia moottorista täytyy kääntää joka iteraatiossa uudestaan [BiS02] – esimerkiksi Unity-pelimottoorin toteutus perustuu tällaiseen ratkaisuun [Uni15].

Koostaminen tuo tietenkin mukanaan myös omat haasteensa: koska komponenttilistat on luotava suoritusaikana, ei käännösaikana voida vielä välttämättä olla varmoja, mitä komponentteja kullakin pelioliolla tulee olemaan. Tämä saattaa olla erityisen vaarallista, jos jokin komponentti on riippuvainen toisesta: kriittisen komponentin puuttuminen ei välttämättä näy käännösvaiheessa ja saattaa estää tärkeitä ominaisuuksia toimimasta [PaE09]. Esimerkiksi mikäli pelihahmon halutaan liikkuvan fysiikkamoottorin sääntöjen mukaan ja kimpoilevan muista kappaleista, siihen täytyy liittää fysiikkakomponentti. Fysiikkakomponentti vaatii kuitenkin toimiakseen ainakin kaksi muuta komponenttia: peliolion sijainnin pelimaailmassa määrittelevän sijaintikomponentin sekä fysiikkakappaleen muodon määrittelevän törmäyshahmokomponentin. Ilman jälkimmäistä fysiikkamoottori kykenee ehkä liikuttamaan oliota maailmassa, mutta ei havaitsemaan törmäyksiä, ja ilman ensimmäistä fysiikkakomponentti ei todennäköisesti pysty suorittamaan lainkaan järkevää laskentaa.

2.4 Rinnakkaistamisstrategiat

Monimutkaisten järjestelmien kuten pelimoottoreiden rinnakkaistaminen voi olla työlästä: tehokkaan rinnakkaistoteutuksen saavuttamiseksi on löydettävä järkevä työnjako ja huolehdittava, että jaettu data säilyy eheänä. Pelimoottorien rinnakkaistamisessa on kaksi yleistä peruslähestymistapaa, joita käytetään usein yhdessä: tehtäväpohjainen ja tietopohjainen rinnakkaistaminen. Tässä luvussa käydään läpi molemmat tekniikat sekä esitellään GPGPU-ohjelmointia, eli yleiskäyttöistä grafiikkasuoritinohjelmointia, jota todennäköisesti käytetään tulevaisuudessa laajalti rinnakkaistamisen apuna.

Tehtäväpohjainen rinnakkaistaminen

Koska pelimoottorit koostuvat moduuleista, intuitiivisin ja yksinkertaisin tapa rinnakkaistaa moottori on ajaa eri moduuleita eri säikeissä. Tätä kutsutaan tehtäväpohjaiseksi rinnakkaisuudeksi (task parallelism), koska tietynlaiset tehtävät suoritetaan aina samassa säikeessä [TBN06]. Menetelmä myös helpottaa eri moduulien suorittamista eri aikataulussa, sillä jokainen erillinen moduulisäie voi suorittaa omaa päivityssilmukkaansa, joka on riippumaton varsinaisen pelisilmukan suoritusnopeudesta [DaW11]. Esimerkiksi fysiikkamoottorin voi määrätä päivittämään tilanteensa tasan 60 kertaa sekunnissa muun pelimoottorin ruudunpäivitystaajuudesta riippumatta, mistä on hyötyä, jos halutaan sen toimivan aina deterministisesti, vaikka järjestelmä olisi kovan kuormituksen alla.

Helppo ja hyödyllinen esimerkki tehtäväpohjaisesta rinnakkaistamisesta on resurssien lataaminen. Se voidaan toteuttaa enimmäkseen riippumatta muusta moottorista, sillä se on pitkäkestoinen operaatio, jota voidaan suorittaa pelin taustalla, kunnes ladattuja resursseja todella tarvitaan [TBN06]. Säikeiden välistä synkronointia tarvitaan siis vasta, kun resurssi pitää tuoda pelimaailmaan. Äänentoisto- ja piirtomoduulit puolestaan rinnakkaistuvat hyvin, koska ne ottavat vastaan syötteitä pelimaailman tilaa hallinnoivilta moduuleilta, mutta eivät itse muuta pelin tilaa [TBN06] – tällöin seuraavan ruudunpäivityksen pelitilaa voidaan laskea rinnakkain samaan aikaan, kun ääni- ja grafiikkakomponentit vielä käsittelevät edellistä ruudunpäivitystä. Sen sijaan pelimaailman tilaan vaikuttavat moduulit kuten fysiikkamoottori tai tekoäly vaativat monimutkaisempaa rinnakkaisuudenhallintaa, sillä ne voivat muuttaa rinnakkain samo-

jen peliolioiden tilaa [TBN06].

Puhtaan tehtäväpohjaisen rinnakkaistamisen ongelmaksi nousee helposti skaalautuvuus sekä laskentaresurssien tehokas hyödyntäminen. Eri moduuleilla on erilaisia tehovaatimuksia, ja mikäli verrattain kevyelle moduulille kuten esimerkiksi äänimoottorille antaa käyttöön kokonaisen prosessoriytimen, ydin kuluttaa suuren osan ajastaan odottaen uusia syötteitä muilta moduuleilta tai muiden säikeiden laskennan valmistumista. Lisäksi, mikäli järjestelmässä olevien prosessoriydinten määrä on suurempi kuin rinnakkaistettujen moduulien, ei kaikille ytimille enää riitä laskettavaa eikä moottori enää skaalaudu suuremmalle määrälle ytimiä.

Tietopohjainen rinnakkaistaminen

Tehtäväpohjaisen rinnakkaisuuden lisäksi pelimoottoreissa hyödynnetään paljon tietopohjaista rinnakkaisuutta (data parallelism), jossa samaa algoritmia ajetaan rinnakkain useilla eri syötejoukoilla [TBN06]. Hyvä esimerkki tietopohjaisesta rinnakkaistamisesta on grafiikan piirtäminen: jokaisen kuvapisteen väriarvo lasketaan samalla tavalla, mutta laskennan lopputulos vaihtelee kuvapisteen esittämän pelimaailman pisteen sisällön – siis piirtoalgoritmin syötteen – perusteella. Piirtoalgoritmien hyvän rinnakkaistuvuuden vuoksi grafiikkasuorittimet sisältävätkin nykyään satoja tai jopa tuhansia ytimiä. Toinen yleinen käyttökohde ovat laskennallisesti raskaat laskentasilmukat, joita löytyy pelimoottoreissa esimerkiksi fysiikkamoottoreista [DaW11]. Realistiselta näyttävän fysiikan simuloiminen vaatii niin paljon laskentatehoja, että simuloitavien fysiikkaolioiden määrän noustessa kymmeniin tai jopa satoihin yksittäinen suoritinydin ei usein pysty mitenkään käsittelemään niitä kaikkia sekunnin kuudeskymmenesosassa. Kokonainen fysiikkamoottorin simulaatioaskel voidaan kuitenkin helposti jakaa pieniksi rinnakkain suoritettaviksi alitehtäviksi, joissa päivitetään vain yhden fysiikkaolion tila.

Uusien säikeiden luomisen raskauden vuoksi tietopohjaiseen rinnakkaistamiseen käytetään usein jonkinlaista työjärjestelmää (task system/job system) [TBN06] ja säieallasta (thread pool). Tämä tarkoittaa, että pelimoottorin komponentit kapseloivat tarvitsemansa laskennan toisistaan riippumattomiksi töiksi ja asettavat ne jonoon tai johonkin muuhun tietorakenteeseen, josta työntekijäsäikeet käyvät hakemassa lisää laskettavaa aina saadessaan nykyisen työnsä valmiiksi.

Tietopohjainen rinnakkaistaminen välttää tehtäväpohjaisen rinnakkaistamisen skaalautu-

vuusongelmat, sillä säikeitä ja ytimiä voidaan periaatteessa hyödyntää yhtä monta kuin rinnakkaistettavalla moduulilla tai algoritmilla on syötteitä: esimerkiksi fysiikkamoottorin tapauksessa jokaista pelimaailman fysiikkaoliota voitaisiin simuloida omalla ytimellään, mikäli ytimiä olisi riittävästi. Toisin sanoen suoritinytimien määrän kasvattaminen lisää teoriassa suorituskykyä aina siihen asti, että ydinten määrä alkaa lähestyä itsenäisten peliolioiden määrää. Käytännössä kuitenkin työn jakaminen prosessoriytimille vaatii aina lisälaskentaa, joten tekniikasta saadaan hyötyä vain, jos työtehtävät ovat tarpeeksi suuria [LaG05]. Liian kevyet tehtävät johtavat siihen, että moottori kuluttaa enemmän aikaa rinnakkaisuuden hallintaan kuin pelitilan laskemiseen. Tästä periaatteesta käytetään nimeä hienojakoisuus (granularity): mitä pienempiä työyksiköt ovat, sitä tasaisemmin työ jakautuu eri prosessoriydinten välille, mutta samalla rinnakkaistamisesta saatava lisäteho pienenee [LaG05]. Tämän vuoksi tietopohjainen rinnakkaistaminen sopii parhaiten hyvin raskaisiin, monimutkaisia komponentteja käyttäviin peleihin: laskennan vaativuuden lisääntyessä rinnakkaisuudenhallinnan suhteellinen osuus tehdystä työstä vähenee ja rinnakkaisuudesta saatava suorituskykyetu kasvaa [LaG05].

GPGPU

Eräs tietopohjaiseen rinnakkaistamiseen läheisesti liittyvä tekniikka on SIMD (single instruction, multiple data) [TBN06], jossa useat eri syötedataa käsittelevät prosessoriytimet jakavat yhteisen ohjelmalaskurin, eli suorittavat samaa koodia rinnakkain, käsky käskyltä, täysin samanaikaisesti. Tekniikalla voidaan laskea tehokkaasti esimerkiksi matriisilaskuja: jokaiselle ytimelle annetaan tehtäväksi laskea tulosmatriisin yhden eri alkion arvo, jolloin koko laskutoimitukseen kuluu vain yhden alkion laskemiseen kuluva aika, olettaen että prosessoriytimiä on vähintään yhtä paljon kuin tulosmatriisin alkioita. Tekniikkaa sovelletaan nykyisin laajalti erityisesti grafiikkasuorittimissa niiden suuren ydinmäärän vuoksi: ydinten koordinointi veisi paljon laskentatehoja ja muistin kaistanleveyttä, mikäli jokainen suorittaisi eri ohjelmaa tai olisi saman ohjelman eri vaiheessa.

SIMD:n merkitys tulee todennäköisesti kasvamaan tulevaisuudessa GPGPU-ohjelmoinnin (General-Purpose Computing on Graphics Processing Units) myötä. GPGPU:n avulla voidaan valjastaa näytönohjaimet suorittamaan yleisiä, grafiikkaan liittymättömiä laskutoimituksia [ZaM07], ja sitä tukevat kaikki nykyisen kahdeksannen sukupolven

pelikonsolit. GPGPU:lla voidaan siirtää tiettyjä raskaita operaatioita kuten vaikkapa pelimoottorin fysiikkalaskelmat näytönohjaimelle, joka kykenee suorittamaan ne tehokkaammin [ZaM07]. Varjopuolena GPGPU-ohjelmointi on perinteistä ohjelmointia hankalampaa; esimerkiksi juuri SIMD:n vuoksi GPGPU-ohjelmissa tulee välttää suorituksen haarautumista, sillä jaetun ohjelmalaskurin vuoksi osan ytimistä suorittaessa yhtä ohjelman haaraa kaikki muut joutuvat odottamaan oman haaransa tulemista suoritusvuoroon.

Pelimoottori on suurikokoinen ohjelmistokehys, jonka päälle useimmat nykypelit rakennetaan. Koska pelit kattavat hyvin monimuotoisen kirjon, pelimoottoreissa painotetaan yleensä modulaarista rakennetta sekä ohjelmistoarkkitehtuurin että yksittäisten peliolioiden mittakaavassa. Jo tämän vuoksi moottoreiden toteutus on monimutkaista, mutta lisäksi myös laitteiston laskentaresurssien hyödyntäminen mahdollisimman tehokkaasti on kriittistä, mikä nykyalustoilla tarkoittaa, että kehitystyössä on otettava huomioon alusta alkaen myös rinnakkaisuus. Useimmissa pelimoottoreissa rinnakkaistaminen on toteutettu yhdistelemällä kahta rinnakkaistamistekniikkaa: tehtäväpohjaista rinnakkaistamista, jossa eri moottorimoduulit jaetaan eri säikeiden vastuulle, sekä tietopohjaista rinnakkaistamista, jossa komponenttien sisällä hajautetaan samankaltaisten tehtävien laskentaa eri säikeille. Tulevaisuudessa on todennäköisesti nousemassa tärkeäksi myös GPGPU-ohjelmointi, jossa helpotetaan keskusyksikön laskentataakkaa siirtämällä osa sen laskennasta massiivisesti rinnakkaiselle grafiikkasuorittimelle. Rinnakkaistamisstrategiasta riippumatta moottorin rinnakkaistoteutus ei kuitenkaan useimmiten ole triviaalia, sillä rinnakkaisuus voi aiheuttaa sekä yleisiä että sovellusalueelle ominaisia ongelmia.

3 Pelimoottorien rinnakkaistamisongelmat

Ohjelmistojen rinnakkaistaminen on harvoin ongelmatonta, eivätkä pelimoottorit ole poikkeus. Tässä luvussa tarkastellaan joitakin pelimoottoria rinnakkaistettaessa mahdollisesti esiintyviä toteutusongelmia ja etsitään niihin ratkaisuja.

3.1 Yleiset rinnakkaistamisongelmat

Pelimoottoreita rinnakkaistaessa voidaan kohdata tietenkin samoja ongelmia kuin minkä tahansa muunkin ohjelman rinnakkaistamisessa. Esimerkiksi jos useasta säikeestä käsitellään yhteiskäytössä olevan muuttujan tilaa samanaikaisesti ilman jonkinlaista lukkoa, semaforia tai vastaavaa rinnakkaisuudenhallintamekanismia, voi muodostua kilpatilanne, jossa ohjelmisto toimii väärin, koska jonkin säikeen tekemät muutokset eivät jää voimaan [BAM06, s.19-20]. Jos usea säie pyrkii varaamaan samoja resursseja, ne voivat lukkiutua, jolloin koko ohjelman suoritus saattaa pysähtyä säikeiden odottaessa ikuisesti suoritusvuoroa toisiltaan [BAM06, s.57-58]. Näiden ja muiden rinnakkaisohjelmoinnin ongelmien löytäminen ja korjaaminen on hankalaa, koska rinnakkaisuuden vuoksi komentojen suoritusaikainen järjestys on epädeterminististinen. Ongelmatilanteen toistaminen voi olla vaikeaa tai jopa käytännössä mahdotonta, jos se aiheutuu jostakin hyvin epätodennäköisestä suoritusketjusta [BAM06, s.21]. Koska rinnakkaisohjelmointivirheiden löytäminen ja rinnakkaisen suorituksen ajattelun sisäistäminen on vaikeaa, rinnakkaisohjelmointia pidetään eräänä hankalimmista ohjelmoinnin ongelmakentistä.

Pelimoottoreiden toteutus on jo ohjelmiston koon vuoksi vaikeaa ja yllä mainitut ongelmat hankaloittavat sitä entisestään. Lisäksi käyttäjät törmäävät harvinaisiinkin rinnakkaisuusvirheisiin nopeasti yksittäisten pelien saavuttaessa nykyään jopa kymmeniä miljoonia loppukäyttäjiä, niiden pohjana toimivien pelimoottoreiden levinneisyydestä puhumattakaan. Toteutusvaikeuden ja vaadittavan laatutason vuoksi moottorin rinnakkaistaminen saattaa osoittautua hyvin kalliiksi.

3.2 Vuorovaikutusketjut

Pelimoottorin skaalautuvuutta rajoittavaksi tekijäksi saattavat nousta pelimoottorikomponenttien vuorovaikutusten muodostamat ketjut. Ennen kuin moottori voi ryhtyä laskemaan seuraavaa pelisilmukan suorituskertaa, on sen laskettava pelimaailman ko-

konaistilanne nykyisen suorituskerran lopussa, minkä vuoksi sen on ratkaistava kaikki komponenttien ja moduulien tuottamat tehtävät [TBN06]: liikkeessä oleville kappaleille on laskettava uusi sijainti fysiikkamoottorin sääntöjen mukaan, tekoälykomponenttien täytyy päättää, mitä pelihahmot tekevät seuraavaksi, ja niin edelleen. Peliolioiden ollessa vuorovaikutuksessa toistensa kanssa nämä tehtävät voivat kuitenkin poikia lisää tehtäviä. Pelimaailmassa voi esimerkiksi olla raskas kuula, joka vierii kohti jonossa olevia laatikoita. Kuulan sijainnin päivittävä fysiikkamoottoritehtävä liikuttaa kuulaa niin, että sen törmäyshahmo ja ensimmäisen laatikon törmäyshahmo ovat päällekkäin – siis kuula törmää laatikkoon. Törmäyksen seurauksena moottorin on laskettava uudet sijainnit sekä kuulalle että laatikolle. Ensimmäisen laatikon liike saa sen kuitenkin törmäämään toiseen, jolloin sillekin on laskettava uusi sijainti, joka saa sen törmäämään kolmanteen ja niin edelleen [MiI07, sivut 388-390].

Tehtävien laskeminen saattaa siis aiheuttaa peliolioiden ja moottorimoduulien välisten vuorovaikutusten ketjuja [TBN06]. Tämä on rinnakkaisuuden kannalta hankalaa, sillä edellisen esimerkin tapauksessa jonon viimeisen laatikon liikettä ei voida käsitellä rinnakkain jonon aikaisempien laatikoiden liikkeen kanssa – vaikka kaikkien laatikoiden liikkeet on käsiteltävä ennen seuraavaa simulaatioaskelta, viimeisen laatikon liikkeitä ei mitenkään voida simuloida ennen kuin on laskettu, että toiseksi viimeinen laatikko tönäisee sitä. Toisin sanoen vuorovaikutusketjut ovat luonnostaan sarjallisia [TBN06]. Ketjut rajoittavat skaalautuvuutta, sillä vaikka käytössä olisi ääretön määrä laskentaytimiä, ruudunpäivityksen laskentaan kuluu vähintään pisimmän vuorovaikutusketjun sarjalliseen laskemiseen vaadittava aika, ja osa ytimistä saattaa joutua odottamaan työttömänä seuraavan simulaatioaskeleen alkua sillä välin, kun yksi ydin ratkoo erityisen pitkää ketjua.

Vuorovaikutukset voivat myös levitä muihin pelimoottorin moduuleihin ja pelioloihin; kuulan ja laatikon törmäys voi saada aikaan äänen, mikä tarkoittaa lisää tekemistä pelin äänimoottorille, ja lähistöllä oleva vihollinen saattaa ”kuulla” törmäyksen ja reagoida siihen, jolloin tekoälykomponentin on laskettava sille uusi käyttäytymistapa. Tämä ei välttämättä hidasta laskentaa, mikäli eri moduulit käyttävät eri päivityssilmukoita: esimerkiksi fysiikkamoottorit käyttävät deterministisyyden saavuttamiseksi usein omaa, muun moottorin pelisilmukasta erillistä silmukkaansa, jolloin vaikkapa tekoälykomponentille aiheutettu lisätyö ei estä fysiikkamoottoria aloittamasta seuraavan

ruudunpäivityksen käsittelyä. Jos moduulit ovat kuitenkin saman pelisilmukan alaisia, niiden keskinäisten vuorovaikutusten ketjut kykenevät hidastamaan laskentaa siinä missä moduulien sisäisetkin.

Mahdollinen lievitys ongelmaan on pitkiä ketjuja synnyttävien tehtävien priorisointi; mikäli pitkien ketjujen laskeminen aloitetaan, kun moottorilla on vielä paljon rinnakkain laskettavia tehtäviä, voidaan koko laskentakapasiteettia hyödyntää mahdollisimman pitkään, ennen kuin jäljelle jää vain sarjallisesti laskettavia tehtäviä. Pitkiä ketjuja aiheuttavia tehtäviä on kuitenkin hankala tunnistaa ennalta tehokkaasti [TBN06]. Yksinkertaisempi ratkaisu on katkaista pitkät ketjut ja siirtää niiden loppuun laskeminen seuraavaan ruudunpäivitykseen. Tämä luo kuitenkin peliolioiden vuorovaikutuksiin viiveitä, joilla voi olla negatiivinen vaikutus pelaajan pelielämykseen etenkin tarkkaa ajoitusta vaativissa peleissä.

3.3 Tilan synkronointi

Toteutustavasta riippumatta kaikissa rinnakkaisissa pelimoottoreissa eri säikeiden suoritustila täytyy ennen pitkää synkronoida: pelitilanteen on oltava yksikäsitteinen, ennen kuin sen voi esittää pelaajalle. Mikäli useista säikeistä voidaan muuttaa samojen muuttujien tilaa, järjestelmän kokonaistilan saa selville vain yhdistämällä kaikkien säikeiden tekemät muutokset [AnJ09]. Monissa rinnakkaissovelluksissa tämä tehdään yksinkertaisesti suojaamalla jaetut muuttujat lukoilla ja antamalla säikeiden käsitellä niitä yksi kerrallaan, mutta mikäli samaa muuttujaa käsitellään usein, tämä saattaa hidastaa suoritusta säikeiden jäädessä odottamaan omaa vuoroaan. Peleissä tämä on erityisen haitallista, sillä suorituskyky on yleensä erittäin tärkeää. Eräs mahdollinen ratkaisu on antaa säikeiden tehdä muutoksensa paikallisiin muuttujiin ja synkronoida muutokset tietyin väliajoin – peleissä usein intuitiivisesti kerran ruudunpäivityksen aikana [AnJ09].

Synkronointi voidaan suorittaa pääsääntöisesti yhdessä kahdesta suoritustilasta, jotka määrittelevät, milloin eri pelimoottorikomponenttien tila synkronoidaan ja milloin moottori jää odottamaan niiden laskennan valmistumista [AnJ09]. Lukittu askel -tilassa kaikki pelimoottorin moduulit suorittavat laskentansa loppuun, ennen kuin moottori aloittaa seuraavan ruudunpäivityksen laskemisen [AnJ09]. Tällöin voi aiheutua ongelmia, jos yksittäisen moduulin laskenta kestää paljon muita kauemmin, sillä seuraavaa ruutua ei päästä laskemaan, ennen kuin kaikki moduulit ovat valmiina. Mikäli kyseinen

pelimoottorimoduuli itsessään ei ole rinnakkaistettu, moottorin suoritus palautuu tällöin sarjalliseksi muiden säikeiden jäädessä odottamaan yhden säikeen laskennan valmistumista. Ongelmaa voidaan lievittää jakamalla raskaan moduulin laskenta osiin, jotka suoritetaan eri ruudunpäivitysten aikana: esimerkiksi tekoäly voi laskea ensimmäisessä ruudussa karkean tavoitteen ja seuraavassa päätellä tarkemmin, miten se saavutetaan. Vapaa askel -tilassa moduuli voi puolestaan varata usean ruudunpäivityksen laskennalleen, jolloin moottori ei jää odottamaan sitä heti vaan vasta useamman pelisilmukan suorituskerran jälkeen [AnJ09]. Tällöin osa moduuleista synkronoidaan esimerkiksi vain joka toisen ruudunpäivityksen yhteydessä, eikä niiden sisäisen tilan tarvitse olla ristiriidaton näiden välisissä synkronointipisteissä. Vapaa askel -tila on kuitenkin vaikeampi toteuttaa ja voi vaatia enemmän toisteista dataa, sillä järjestelmän tila saattaa muuttua raskaiden moduulien laskennan aikana, jolloin niiden suorittama laskenta saattaa perustua vanhentuneeseen dataan, joka voi aiheuttaa virhetilanteita [AnJ09]. Lisäksi jotkin moduulit voivat myös toimia täysin ruudunpäivityksistä erillään: jos moottori vaikkapa lataa taustalla seuraavan pelialueen 3D-malleja ja tekstuureita, ei tällä ole mitään vaikutusta pelin reaaliaikaiseen tilaan, joten se saa valmistua omalla ajallaan [AnJ09].

Synkronointia voi edelleen vaikeuttaa se, että osa pelimoottorin moduuleista saattaa vaatia tiettyä ruudunpäivitystaajuutta toimiakseen; esimerkiksi monet fysiikkamoottorit suorittavat päivityssilmukkaansa muun pelin ruudunpäivitystaajuudesta riippumatta tasan 30 tai 60 kertaa sekunnissa, jotta simulaatio pysyy deterministisenä. Käytännössä tämä voidaan toteuttaa usealla eri tavalla [GrJ09, s.673-675]. Moduulille voidaan luoda oma pelisilmukka, jota suoritetaan erillisessä säikeessä. Tämä mahdollistaa moduulin oman silmukan päivittämisen periaatteessa täysin pääsäikeen ruudunpäivitysnopeudesta riippumatta, mutta vaatii toimiakseen pää- ja apusäikeen välistä synkronointia, jotta laskennassa käytetään varmasti ajantasaista tietoa [GrJ09, s. 674]. Muutoin pelimoottorin pääsäie saattaa esimerkiksi lukea joidenkin peliolioiden sijainnit ennen fysiikkamoottorin silmukan suoritusta ja joidenkin sijainnit sen jälkeen, jolloin pelaajalle esitettävä pelimaailman tila on ristiriitainen. Toinen vaihtoehto on suorittaa moottorin pelisilmukkaa normaalisti, kunnes moduulin tila on päivitettävä. Tällöin pelisilmukan suoritus pysäytetään väliaikaisesti ja moottori luo useita säikeitä, jotka laskevat moduulin päivityksen mahdollisimman nopeasti. Kun päivitys on suoritettu, pelisilmukan suoritus jatkuu [GrJ09, s. 674-675]. Tämä kuitenkin vaatii toimiakseen tehokkaasti, että moduu-

lin sisäinen laskenta rinnakkaistuu hyvin – muutoin osa laitteiston suoritinytimistä joutuu odottamaan moduulin laskennan päättymistä toimettomina. Jos moduulin laskenta on jaettavissa erillisiin vaiheisiin ja moottori on rinnakkaistettu työjärjestelmän (ks. luku 2.4, Tietopohjainen rinnakkaistaminen) avulla, on olemassa myös kolmas vaihtoehto: moduulin laskentavaiheet voidaan toteuttaa työtehtävinä, jotka lisätään työjärjestelmään, kun moduulin tilaa tarvitsee päivittää [GrJ09, s. 675]. Tällöin synkronointi helpottuu, koska vaiheiden tarkat suoritusajankohdat voidaan valita niin, että pääsäikeellä on samanaikaisesti jotakin järkevää mutta riippumatonta laskettavaa.

Mikäli samaan muuttujaan kohdistuu useita rinnakkaisia muutoksia eri säikeistä yhden ruudunpäivityksen aikana, on valittava, miten muutokset yhdistetään. Tähän käytetään joskus erityistä tilanhallintamoduulia, jonka tehtävänä on ottaa muilta moduuleilta komponenteilta vastaan ilmoituksia jaetun muistin tilan muutoksista ja tiedottaa niistä edelleen muille kyseisen muuttujan tilan muutoksista kiinnostuneille moduuleille ja komponenteille [AnJ09]. Varsinainen valintastrategia riippuu yleensä muuttujasta – esimerkiksi hahmon sijaintimatriisiin kohdistuvissa muutoksissa saattaa olla järkevää jättää voimaan vain yksi muutos valittuna muutoksen tekijän prioriteetin perusteella, sillä peräkkäiset muutokset saattavat johtaa eri lopputuloksiin suoritusjärjestyksestä riippuen [AnJ09]. Sen sijaan vaikkapa pelaajan pistemäärän tapauksessa kannattaa muutokset tehdä suhteellisina, eli absoluuttisen arvon sijaan lähettää tilanhallinnalle muuttujan tilan muutoksen määrä, koska yleensä järjestyksellä ei ole väliä, ja esimerkiksi etenkin kilpailullisissa moninpeleissä on pelin lopputuloksen kannalta tärkeää, että pistelaskuri toimii tarkasti eikä jätä yhtään pistettä huomiotta.

Rinnakkaisohjelmointi tuo ohjelmistokehitykseen aina omat haasteensa: rinnakkaisuus vaikeuttaa testausta ja ohjelmointivirheiden korjausta sekä vaatii kehittäjältä rinnakkaisen suorituksen ymmärtämistä. Pelimoottoria rinnakkaistaessa on myös otettava huomioon sovellusalueelle ominaisia ongelmakohtia, kuten peliolioiden monimutkaisten vuorovaikutusten aiheuttamat sarjalliset ketjut. Myös peliolioiden tilojen synkronointi säikeiden välillä useita kymmeniä kertoja sekunnissa on hankalaa: jos saman peliolion tilaa yritetään muuttaa eri säikeistä, on pidettävä huoli, että muutokset yhdistetään järkevällä tavalla ennen lopputuloksen esittämistä käyttäjälle, ja mikäli tämä tapahtuu ainakin kerran jokaisessa ruudunpäivityksessä, on prosessi kyettävä toistamaan vähintään 30 kertaa jokaisen sekunnin aikana. Pelimoottoreiden rinnakkaistamisessa

käytetään kuitenkin usein perinteisiä rinnakkaistamistekniikoita kuten lukkoja, joten toteutuksen hankaluutta voisi mahdollisesti lievittää hyödyntämällä moderneja, korkeamman abstraktiotason lähestymistapoja.

4 Rinnakkaisohjelmointi ja aktorit

Aktorit ovat rinnakkaisohjelmointitekniikka, joka perustuu itsenäisiin, asynkronisen viestinvälityksen avulla toistensa kanssa keskusteleviin olioihin eli aktoreihin, joita voidaan suorittaa rinnakkain eri säikeissä. Aktorimalli pyrkii ratkaisemaan tai lievittämään monia perinteisten rinnakkaistamismenetelmien ongelmia välttämällä jaetun muistin käyttöä ja eristämällä kaikki muuttujat osaksi jonkin aktorin paikallista tilaa. Tässä luvussa tarkastellaan ensin C++11-kielen sisältämiä perinteisiä rinnakkaistamismekanismeja ja sitten aktorimallin historiaa, hyötyjä ja toimintaperiaatteita.

4.1 C++11:n rinnakkaisuudenhallintamekanismit

Ennen aktorimallin esittelemistä on syytä valottaa myös perinteisiä, lukkopohjaisia rinnakkaisuudenhallintamekanismeja, joita käytetään useimpien nykyisten pelimoottorien rinnakkaisuudenhallinnan pohjana. Nämä mekanismit ovat saatavilla monien eri kirjastojen kautta, mutta tässä aliluvussa tarkastellaan erityisesti keskeisimpiä C++11-kieleen sisäänrakennetuista rinnakkaisuudenhallinnan työkaluista, koska tätä tutkielmaa varten toteutettu Stage 11 -moottori (ks. luku 6.1, Stage 11) rinnakkaistettiin juuri C++11:n sisäänrakennettujen rinnakkaisuudenhallintamekanismien avulla.

C++11:n rinnakkaisuudenhallinnan ytimessä on std::thread-luokka, joka mallintaa yksittäisen laskentasäikeen [ISO12, s. 1117-1123]. C++11 mahdollistaa uuden säikeen luomisen yksinkertaisesti luomalla std::thread-olion, jonka konstruktorille annetaan parametreiksi funktio, jota säikeen halutaan suorittavan sekä kyseiselle funktiolle annettavat parametrit. Kun säie on luotu, se aloittaa laskentansa automaattisesti ja pysähtyy päästyään suorittamansa funktion loppuun. Tärkein säieolion itse tarjoamista metodeista on join, jota kutsuva säie jää odottamaan säieolion mallintaman laskentasäikeen suorituksen päättymistä.

Kun ohjelma sisältää useita rinnakkaisia laskentasäikeitä jotka käyttävät samoja jaetussa muistissa olevia muuttujia, saatetaan törmätä kilpatilanteisiin: esimerkiksi jos kaksi eri säiettä yrittää samanaikaisesti kasvattaa saman muuttujan arvoa, saattaa vain toisen tekemä muutos jäädä voimaan. Tämä estetään kontrolloimalla lukkojen avulla pääsyä niin kutsutulle kriittiselle alueelle (critical section), eli siihen ohjelman osaan, jossa eri säikeet saattaisivat häiritä toistensa laskentaa. Jokaisen samaa muuttujaa käyttävän säikeen

on otettava haltuunsa sama lukko ennen pääsyä omalle kriittiselle alueelleen, mutta vain yksi säie voi pitää lukkoa hallussaan kerrallaan. Toisin sanoen mikäli säie haluaa päästä kriittiselle alueelleen toisen säikeen pitäessä lukkoa hallussaan, se joutuu odottamaan, kunnes toinen säie pääsee oman kriittisen alueensa loppuun ja vapauttaa lukon. Näin varmistutaan, että vain yksi säie pääsee kerrallaan käyttämään saman muistialueen sisältöä.

C++11:n yksinkertaisin lukkotyyppi on std::mutex [ISO12, s.1123-1125]. Kun yksi säie on ottanut std::mutex-olion haltuunsa kutsumalla sen lock-metodia, muut saman olion lock-metodia kutsuvat säikeet joutuvat odottamaan, kunnes alkuperäinen säie vapauttaa lukon kutsumalla olion unlock-metodia. Kehittäjän ei kuitenkaan suositella kutsuvan näitä metodeja itse, sillä jos säikeen kriittisellä alueella heitetään poikkeus, saattaa säikeen laskenta keskeytyä ennen unlock-metodin kutsumista, jolloin lukko jää pysyvästi lukituksi ja koko ohjelma saattaa lukkiutua. Ongelma on ratkaistu std::lock_guard -luokan avulla [ISO12, s.1128-1129]. std::lock_guard-oliolle annetaan konstruktoriparametrina std::mutex-olio, jonka se lukitsee koko elinajakseen: lukko otetaan haltuun std::lock_guard-olion konstruktorissa ja vapautetaan sen destruktorissa. Kun funktio luo paikallisen std::lock_guard-olion ennen kriittistä vaihettaan, lukko vapautuu myös virhetilanteissa, sillä suorituksen keskeytyessä poikkeuksen vuoksi kaikkien paikallisten muuttujien sisältämät oliot tuhotaan kutsumalla niiden destruktoreita. std::unique_lock-luokka toimii pitkälti kuten std::lock_guard, mutta mahdollistaa lisäksi lukon varaamisen ja vapauttamisen manuaalisesti [ISO12, s.1129-1133]. Tämä on hyödyllistä siksi, että antamalla std::lock-funktiolle parametrina useita lukitsemattomia std::unique_lock-olioita, ne voidaan lukita kerralla atomisesti, ilman useiden lukkojen samanaikaisen varaamisen yhteydessä normaalisti helposti esiintyvän lukkiutumisen (deadlock) vaaraa.

Pelkän jaetun muistin suojaamisen lisäksi rinnakkaisohjelmoinnissa tarvitaan eri laskentasäikeiden välistä synkronointia – tämä on erityisen tärkeää pelimoottoreissa, joissa tila on yleensä synkronoitava jokaisen ruudunpäivityksen laskennan päätteeksi. Tätä varten C++11 sisältää ehtomuuttujat (condition variables) [ISO12 s.1135-1140], jotka on toteutettu luokassa std::condition_variable. Kun std::condition_variable-olion wait-metodia kutsutaan, kutsuva säie jää odottamaan, kunnes jokin muu säie kutsuu jompaakumpaa saman olion ilmoitusmetodeista – notify_one herättää yhden ilmoitusta odottavan säikeen ja notify_all kaikki. Tätä voidaan hyödyntää esimerkiksi työjärjestelmää käyttävän

pelimoottorin säiealtaan toteutuksessa: pelisilmukan suoritusvaiheen alussa pääsäie lisää järjestelmään joitakin työtehtäviä, herättää työntekijäsäikeet ja kutsuu itse laskennan päättymisestä ilmoittavan ehtomuuttujan wait-metodia. Työntekijäsäikeet suorittavat työtehtäviä niin kauan kuin niitä on tarjolla ja jäävät sitten odottamaan signaalia toiselta ehtomuuttujalta, jonka ilmoitusmetodia kutsutaan aina uusien työtehtävien saapuessa järjestelmään. Vaiheen laskenta on saatu päätökseen, kun järjestelmässä ei enää ole uusia työtehtäviä jäljellä – tällöin kaikki työntekijäsäikeet jäävät odottamaan uusia tehtäviä ja viimeinen herättää pääsäikeen käynnistämään pelisilmukan seuraavan vaiheen.

Edellä mainittujen lukkopohjaisten rinnakkaisuudenhallintamekanismien lisäksi C++11 sisältää myös futuurit (futures) [ISO12, s.1144-1152], jotka mahdollistavat arvojen palauttamisen toisessa säikeessä asynkronisesti suoritettavista funktioista. Futuurit on mallinnettu luokassa std::future, jonka ilmentymän voi luoda muun muassa std::promise-olion get_future-metodilla. Kun toisessa säikeessä suoritettava funktio tuottaa paluuarvon, se voidaan kirjoittaa futuuriin saman std::promise-olion set_value-metodilla. Tämän jälkeen paluuarvo voidaan lukea muista säikeistä std::future-olion get-metodilla – jos metodia kutsutaan ennen futuurin arvon asettamista, kutsuva säie jää odottamaan paluuarvon saapumista.

4.2 Aktorimalli ja sen toteutus Theron-kirjastossa

Aktorimalli on alun perin tekoälyn ja laskennan mallien teorioiden tutkimista varten kehitetty rinnakkaisuusmalli. Mallin loi Carl Hewitt, joka esitteli sen ensi kertaa Peter Bishopin ja Richard Steigerin kanssa kirjoittamassaan artikkelissa *A universal modular ACTOR formalism for artificial intelligence* [HBS73]. Kiinnostus aktoreita kohtaan on jälleen herännyt rinnakkaisohjelmoinnin yleistymisen myötä ja nykyään malli sisältyy joihinkin moderneihin ohjelmointikieliin, kuten Erlang [VeR09], F# [PeT12] ja Scala [VeR09]. Lisäksi monille muille ohjelmointikielille on saatavilla aktorikirjastoja; koska pelimoottoreiden yleisin toteutuskieli on C++, tässä tutkielmassa käytetään C++-kielen Theron-aktorikirjastoa [MaA14]. C++-kielelle on olemassa muitakin aktorikirjastoja kuten C++ Actor Framework [CAF16], mutta valitettavasti eri kirjastojen suorituskykyä ja ominaisuuksia vertailevaa tutkimusta ei kirjoitushetkellä ollut juurikaan saatavilla, joten Theron valittiin tämän tutkielman aktoriohjelmien pohjaksi hyvän dokumentaationsa perusteella. Jatkossa olisi järkevää kartoittaa C++:n eri aktorikirjastojen tehokkuutta ja

toteutusratkaisuja, jotta saataisiin selville, miten hyvin ne soveltuvat juuri pelimoottorien tarpeisiin. Tässä tutkielmassa esitettävät algoritmit ja toteutusratkaisut ovat joka tapauksessa toteutettavissa millä tahansa aktorimallin periaatteita noudattavalla aktorikirjastolla, ellei erityisesti toisin mainita.

Theronissa määritellään aktoriolio perimällä kirjaston Actor-luokka ja rekisteröimällä yksi tai useampi viestinkäsittelijämetodi [MaA14]. Listauksessa 4.1 määritellään esittelijäaktori, joka vastaa saamiinsa merkkijonotyyppisiin viesteihin tulostamalla konsoliin tervehdyksen ja lähettämällä takaisin oman nimensä. Aktorille voidaan määritellä uusia viestinkäsittelijämetodeja Actor-luokan tarjoamalla RegisterHandler-metodilla – Theron ei vaadi, että metodit tulisi rekisteröidä aktoria luodessa, vaan sallii käsittelijöiden lisäämisen ja poistamisen missä tahansa aktorin elinkaaren vaiheessa [MaA14]. Tästä on hyötyä erityisesti, mikäli halutaan muuttaa aktorin käyttäytymistä suorituksen eri vaiheissa. Käsittelijöiden suoritusaikainen lisääminen ja poistaminen ovat Theron-kirjaston erikoisominaisuus, joka ei välttämättä löydy muista kirjastoista, mutta sitä voidaan teoriassa simuloida missä tahansa aktorijärjestelmässä käyttämällä oletusviestinkäsittelijää, joka valitsee viestin varsinaisesti käsittelevän metodin aktorin sisäisten muuttujien tilan perusteella.

Itse viestinkäsittelijämetodit ovat kaksiparametrisia metodeja, joiden ensimmäinen para-

```
Class Introducer : public Theron::Actor{
public:
        //Konstruktori
        Introducer(Theron::Framework& fw, std::string name) :
             Theron::Actor(fw), name(name) {
                 //Rekisteröidään viestinkäsittelijämetodi
                 RegisterHandler(this, &Introducer::introduce);
        }
        /*Epäoleelliset metodit jätetty pois selkeyden vuoksi
        */
private:
        //Tämän olion 'nimi'
        std::string name;
        //String-tyyppisiä viestejä käsittelevä metodi
        void introduce(const std::string& other,
             const Theron::Address from){
                 //Tulostetaan tervehdys
                 std::cout << "Hi," << other << ",I'm" << name << "!\n";
                 //Lähetetään vastauksena oma nimi
                 Send(name, from);
        }
};
```

Listaus 4.1: Esittelijäaktori

metri on vastaanotettu viesti ja toinen lähettäjän osoite. Kun aktori vastaanottaa viestin, se valitsee suoritettavan metodin viestiparametrin tietotyypin perusteella. Mikäli samalle viestityypille on rekisteröity useita käsittelijöitä, ne suoritetaan kaikki, ja mikäli sama käsittelijä on rekisteröity useasti, se suoritetaan niin monta kertaa kuin se on rekisteröity [MaA14]. Myös useiden käsittelijöiden rekisteröiminen samalle viestityypille on Theron-kirjaston oma ominaisuus, joka ei välttämättä löydy muista kirjastoista, mutta sitä voidaan aina simuloida välittämällä sama viesti useille apumetodeille varsinaisesta viestinkäsittelijästä. Viestien ainoat rajoitteet ovat, että niiden on oltava kopiokonstruoitavissa ja että kerralla voidaan lähettää vain yksi arvo tai olio – jos viestiin halutaan sisällyttää useita parametreja, ne kannattaa lähettää tietueen sisällä. Tämä on hyödyllistä myös siksi, että se vähentää epäselvyyttä, jos samoja tietotyyppejä halutaan käyttää eri konteksteissa. Oletetaan esimerkiksi, että esittelijäaktorin kanssa samassa järjestelmässä toimii toinen aktori, joka vastaanottaa merkkijonoja ja kirjaa ne lokiin. Jos esittelijän ja lokikirjurin osoitteet pääsevät nyt ohjelmointivirheen vuoksi vaihtumaan, alkaa kirjuri kirjaamaan lokiin nimiä ja esittelijä tervehtimään lokimerkintöjä. Jos esittelijän introduce-metodi kuitenkin muutetaan vastaanottamaan Name-tyyppisiä tietueita (jotka sisältävät vain yhden merkkijonon) ja kirjuria muokataan vastaanottamaan Log-tyyppisiä tietueita (sisällöltään identtisiä Name-tietueiden kanssa), huomaa Theron tietotyypistä heti, jos viesti saapuu vääräntyyppiselle aktorille, koska esittelijällä ei ole Log-tyyppisiä eikä kirjurilla Name-tyyppisiä viestejä käsittelevää viestinkäsittelijää. Jos sopivaa viestinkäsittelijämetodia ei löydy, aktorijärjestelmä kutsuu virheenkäsittelijämetodia: Theron-kirjaston oletusvirheenkäsittelijä pysäyttää koko ohjelman suorituksen, mutta ohjelman toteuttaja voi halutessaan korvata sen itse määrittelemällään funktiolla. Aktorien sisältä viestien lähettäminen onnistuu yksinkertaisesti Send-metodilla, jolle annetaan parametreiksi viesti ja vastaanottajan osoite [MaA14].

Aktorien luominen Theronissa on esitelty listauksessa 4.2. Aktoriohjelmaa käynnistettäessä on ensin luotava aktorijärjestelmä, mikä onnistuu Theron-kirjastossa luomalla Theron::Framework-olio [MaA14]. Järjestelmä on vastuussa viestien välittämisestä aktoreille ja pitää yllä säieallasta, jossa aktoreita suoritetaan. Oletuksena työntekijäsäikeitä luodaan Theron-kirjaston järjestelmässä 16 ja se pyrkii siirtämään aktoreita säikeiden välillä siten, että joka ytimelle päätyy yhtä paljon laskettavaa.

Aktorijärjestelmän ulkopuolelta voidaan lähettää viestejä järjestelmän aktoreille järjes-

```
Theron::Framework fw;
Introducer first(fw, "First");
Introducer second(fw, "Second");

if(!fw.Send(std::string("MainThread"),
       second.GetAddress(), first.GetAddress())){
              printf("Failed to send message\n");
}
```

Listaus 4.2: Aktorin luominen

telmäolion Send-metodilla, jolloin on erikseen määriteltävä lähettäjän osoite. Metodi palauttaa arvonaan tiedon siitä, onnistuiko lähettäminen, mikä ei tosin takaa, että vastaanottaja koskaan käsittelee sen, sillä vastaanottaja-aktori saattaa esimerkiksi parhaillaan käsitellä viestiä, joka lopettaa sen suorituksen [MaA14]. Esimerkissä luodaan kaksi esittelijäaktoria, joista ensimmäiselle lähetetään viesti, jonka lähettäjäksi ilmoitetaan toinen. Tällöin ensimmäinen aktori lähettää vastauksensa toiselle, johon toinen vastaa lähettämällä vastauksen takaisin ensimmäiselle – esimerkin aktorit siis esittelevät itsensä vuorotellen toisilleen ikuisesti.

4.3 Aktorien ominaisuudet ja aktoriohjelmointityyli

Aktorimalli tarjoaa perinteisiin rinnakkaisohjelmointitekniikoihin verrattuna useita hyötyjä, jotka helpottavat toteutusta ja auttavat välttämään monia rinnakkaisuuden aiheuttamia ongelmia [OSV08, s.691-692]. Mallissa käsitellään vain aktorien sisäistä tilaa; säikeiden kesken jaettua globaalia tilaa ei aktorimallissa ole, jolloin poissulkemisongelmaa ei tarvitse miettiä. Myös lukkiutumisongelmia voidaan ehkäistä tehokkaasti viestinvälityksen asynkronisuuden avulla: odottaessaan vastausta johonkin lähettämäänsä viestiin aktorin suoritus ei pysähdy, vaan se voi yhä vastata muiden aktorien lähettämiin viesteihin, jolloin lukkiutumisen vaatimaa syklistä odotusta on vaikeampi aiheuttaa. Aktorimalli mahdollistaa myös järjestelmän rinnakkaisresurssien tehokkaan hyödyntämisen ilman, että toteuttajan tarvitsisi miettiä optimointia: järjestelmä jakaa saapuvat viestit eri säikeiden laskettaviksi automaattisesti, eikä kehittäjän tarvitse itse miettiä, jakautuvatko työt eri ytimien välille tasapuolisesti.

Jotta nämä hyödyt voitaisiin saavuttaa, on aktoriohjelmoinnissa kuitenkin noudatettava tiettyjä rajoituksia [OSV08, s.700-704]:

- Aktorin metodeja saa kutsua suoraan vain aktori itse – aktorin ulkopuolelta akto-

rin tarjoamia palveluita saa käyttää vain lähettämällä sille viestejä, koska muutoin aktorin sisäistä tilaa saatetaan muuttaa useista eri säikeistä.

- Aktorit saavat vaikuttaa vain omaan tilaansa, ei jaettuun muistiin. Eri aktoreita voidaan suorittaa eri säikeistä, joten jaetun muistin käyttäminen johtaisi kilpatilanteisiin.
- Aktorit eivät saa pysähtyä odottamaan ulkopuolisen laskennan päättymistä. Jos aktorien viestintä ja vuorovaikutus on synkronista, järjestelmä saattaa lukkiutua aktorien jäädessä odottamaan toisiaan.

Theron ei valvo näiden sääntöjen noudattamista – ohjelmoija voi halutessaan kiertää ne ja ajoittain tämä voikin olla tehokkuussyistä järkevää. Tällöin on kuitenkin syytä olla varovainen, sillä sääntöjen rikkominen johtaa yleensä siihen, että on käytettävä joitakin perinteisistä rinnakkaisuudenhallintamekanismeista, jolloin rinnakkaisuusongelmien välttäminen on vaikeampaa kuin puhtaassa aktorimallissa.

Viimeinen aktorimallin rajoituksista on, että viestiparametrien tulisi olla muuttumattomia (immutable), jotta niiden tilaa ei voida muuttaa samanaikaisesti eri aktoreiden sisältä [OSV08, s.704-705]. Tämä on Theronissa kierretty käyttämällä viesteissä arvoparametreja, joista luodaan eri aktoreille omat kopiot – tällöin parametrin tilan muuttaminen yhdessä aktorissa ei vaikuta muiden saamiin viesteihin. Tämä tietenkin vaatii, että viestiparametrilla on järkevä kopiointisemantiikka, eli esimerkiksi jos parametrilla on kenttänä osoitin, sen osoittamasta arvosta tai oliosta on luotava oma kopio kaikille parametrin kopioille [MaA14]. Theron ei valvo tämän rajoituksen noudattamista sen enempää kuin aikaisempienkaan, vaan oikean kopiointisemantiikan valitseminen jätetään ohjelmoijan vastuulle. Lisäksi suurten, monimutkaisten olioiden kohdalla arvoparametrien käyttö on tehotonta, sillä niiden kopiointi on raskasta – tällaisissa tapauksissa Theronin dokumentaatio kehottaa lähettämään olion sijasta osoittimen siihen ja hallinnoimaan luku- ja kirjoitusoikeuksia synkronointiviestien avulla [MaA14]. Esimerkiksi jos aktori lähettää toiselle osoittimen johonkin sisäiseen tietorakenteeseensa, sen on sitouduttava tekemästä muutoksia tietorakenteen sisältöön, kunnes toinen aktori ilmoittaa viestillä saaneensa rakenteeseen kohdistuvan laskentansa päätökseen.

4.4 Aktorit pelimoottoreissa

Aktoreita ei aikaisemmin ole juurikaan hyödynnetty pelialalla. Kaikkein huomattavim-

missa tapauksissa aktoreita on käytetty jonkin pelin osa-alueen tai siihen liittyvän palvelun kuten moninpelipalvelimien toteutuksessa – esimerkiksi Microsoftin Halo 4 -pelin moninpelitilastoja ylläpitävät palvelimet toteutettiin Orleans-aktorikirjaston avulla [McC15]. Kokonaisten pelien toteuttamista aktorien avulla on kokeiltu joitakin kertoja [KuR13, s.55-58][AtD13], mutta näissä tapauksissa ennestään olemassa oleva peli, joka ei pohjautunut varsinaiseen yleiskäyttöiseen pelimoottoriin, muutettiin aktoripohjaiseksi. Adam Lake ehdottaa kirjassaan *Game Programming Gems 8* rinnakkaisen pelimoottorin toteuttamista aktorien avulla [LaA10, sivut 475-484], mutta kirjassa annettu esimerkki kattaa vain pelimoottorin ytimen eikä ratkaisun mielekkyyttä erityisemmin tutkita tai perustella. Useimmat käytössä olevat rinnakkaiset pelimoottoritoteutukset pohjautuvat siis matalan abstraktiotason rinnakkaisuudenhallintamekanismeihin, kuten semaforeihin tai luku- ja kirjoituslukkoihin, mutta aktoripohjaisella toteutuksella näyttäisi kuitenkin olevan näihin menetelmiin verrattuna joitakin merkittäviä etuja:

- Sekä aktorimalli että pelimoottorit pohjautuvat hyvin samankaltaisiin abstraktioihin. Aktorimallissa itsenäiset aktorit reagoivat toistensa ja järjestelmän lähettämiin viesteihin, kun taas pelimoottoreissa itsenäiset pelioliot päivittävät omaa tilaansa toistensa ja peliympäristön tilan ja tapahtumien perusteella.
- Monissa pelimoottoreissa käytetään viesti- tai tapahtumajärjestelmää [GrJ09, s.773-774], ja luvussa 3.3 kuvattu säikeiden välinen synkronointi on luontevaa toteuttaa sen avulla. Jos rinnakkaistaminen toteutetaan aktoreilla, nämä järjestelmät saadaan lähestulkoon ilmaiseksi hyödyntämällä pohjana aktorijärjestelmän omaa viestinvälitysmekanismia.
- Pelien laitteistoalustojen laskentatehoa kasvatetaan nykyään lisäämällä rinnakkaisten ytimien määrää ja pelit hyödyntävät tätä kasvavaa laskentatehoa enimmäkseen kasvattamalla peliolioiden määrää; ympäristöistä tehdään suurempia ja ne sisältävät yhä enemmän yksityiskohtia. Tämän vuoksi aktoripohjainen rinnakkaistaminen vaikuttaisi skaalautuvan hyvin pelimoottoreiden kohdalla: kun pelioliot toteutetaan aktoreina, niiden tilanpäivityksiä voidaan suorittaa rinnakkain, minkä vuoksi niiden määrän kasvaessa syntyvä lisätyö voidaan jakaa tehokkaasti eri suoritinytimille.
- Aktorimalli helpottaa monien hankalien rinnakkaisuusongelmien kuten poissul-

kemis- ja lukkiutumisongelman välttämistä. Tämän vuoksi aktoripohjaisten ohjelmien kehitys ja ylläpito on usein helpompaa kuin perinteisiä rinnakkaisuudenhallintamekanismeja hyödyntävien ohjelmien. Tämä on erityisen hyödyllistä suurten ja monimutkaisten järjestelmien kuten pelimoottorien toteutuksessa.

Näiden seikkojen vuoksi on hyödyllistä tutkia, miten alusta alkaen aktorien pohjalta toteutettu yleiskäyttöinen pelimoottori toimisi ja minkälaisia vaikutuksia aktoripohjaisuudella käytännössä olisi pelimoottorin käytettävyyteen ja suorituskykyyn.

Aktorimalli on rinnakkaistamistekniikka, joka perustuu itsenäisiin ja rinnakkaisiin, asynkronisen viestinvälityksen avulla kommunikoiviin olioihin. Tekniikkaa hyödyntävän kehittäjän ei tarvitse juurikaan huolehtia monista perinteisempien rinnakkaistamistekniikoiden tuomista ongelmista kuten esimerkiksi poissulkemisongelmasta, mutta hänen on noudatettava tiettyjä rajoituksia, joiden avulla voidaan välttää globaalin tilan käsittely. Jotkin aktorimallin ominaisuudet vaikuttavat myös hyödyllisiltä erityisesti pelimoottorien toteuttamisessa: aktorit ja pelit perustuvat toisiaan muistuttaviin abstraktioihin, aktorit antavat hyvän pohjan tapahtumajärjestelmille, aktorien skaalautuvuus vaikuttaa hyödylliseltä nykypäivän mittakaavaltaan kasvavissa peleissä ja aktoripohjaisuus auttaa välttämään rinnakkaisuusongelmia, jotka voivat olla hyvin hankalia pelimoottorien kaltaisissa monimutkaisissa ohjelmissa. Tämän takia on kiinnostavaa tutkia, miten aktoreiden avulla toteutettu pelimoottori toimisi ja minkälaisiin haasteisiin sen toteutuksessa törmättäisiin.

5 Aktoripohjainen pelimoottoritoteutus

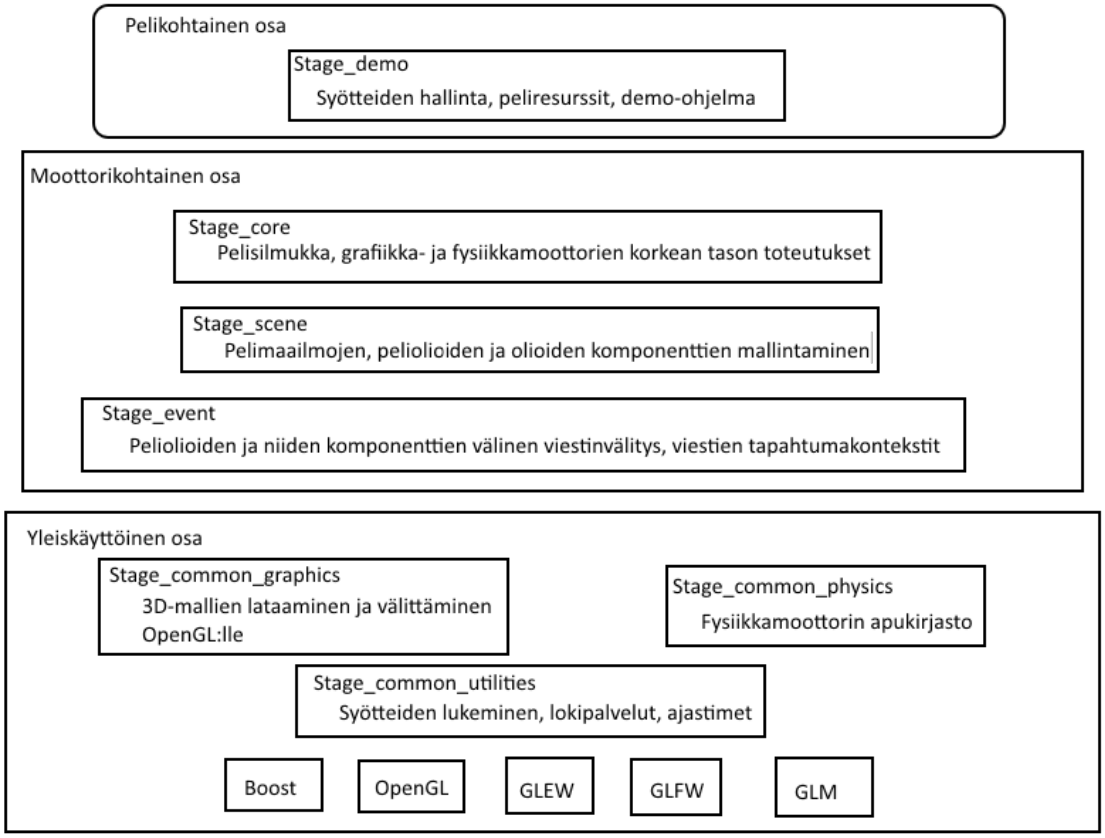
Vaikka aktoreita pelimoottoreissa ei aikaisemmin ole juurikaan tutkittu, aktorijärjestelmillä on tiettyjä ominaisuuksia, jotka vaikuttaisivat hyödyllisiltä rinnakkaista pelimoottoria toteutettaessa (ks. luku 4.3). Jotta saataisiin selville, miten nämä hyödyt toteutuvat käytännössä ja miten hyvin aktoripohjaisuus muuten soveltuu pelimoottorien toteutukseen, tätä tutkielmaa varten kehitettiin yksinkertainen, aktoripohjainen testipelimoottori, Stage, jonka toteutus ja toimintaperiaate esitellään tässä luvussa.

5.1 Pelimoottorin arkkitehtuuri

Stage on tätä tutkielmaa varten toteutettu Theron-kirjastoon pohjautuva rinnakkainen, aktoripohjainen pelimoottori, jonka rakenne on esitelty kuvassa 5.1 ja jonka ohjelmakoodi on liitteissä 1 ja 2. Moottorin pohjana toimii yleiskäyttöinen osa, joka sisältää varsinaisen moottorin käyttämiä palveluita ja kirjastoja – tämä osa on rinnakkaistamaton eikä käytä aktoreita. Yleiskäyttöinen osa sisältää moottorin käyttämät kolmannen osapuolen kirjastot ja API:t: grafiikka-API OpenGL:n [Khr15], sen hallinnointiin käytettävät GLFW- ja GLEW-kirjastot [GLF15][GLE15], GLM-matematiikkakirjaston [GTr15] sekä Boost-apukirjaston [DAR15] C++:lle. Kirjastojen päällä on kolme moduulia, jotka tarjoavat yleistä toiminnallisuutta – koska näitä moduuleja ei ole sidottu Stage-moottorin toteutukseen vaan niiden tarjoamia palveluita voidaan hyödyntää sellaisenaan myös muissa sovelluksissa, niiden nimet alkavat etuliitteellä stage_common.

Stage_common_utilities-moduuli sisältää moottorin käyttämiä aputoimintoja: lokipalvelun toteuttavan Logger-luokan, käyttäjän syötteitä lukevan Input-luokan sekä suorituskyvyn mittaamisen tarkoitetun Timer-luokan. Stage_common_physics-moduuli sisältää peliolioiden törmäysten tunnistamiseen ja fysiikkamoottorin muihin laskelmiin apufunktioita sisältävän Collisions-luokan, sekä törmäyshahmoa mallintavan abstraktin Collider-luokan toteutuksineen. Stage_common_graphics-moduulista puolestaan löytyvät 3D-malleja, pelikameraa ja sävytinohjelmia hallinnoivat luokat sekä 3D-mallien ruudulle piirtämisen mahdollistavana yksinkertaisena grafiikkamoottorina toimiva GraphicsController-luokka.

Näiden moduulien päälle on rakennettu varsinaisen moottorin muodostavat moduulit, jotka hyödyntävät toiminnassaan aktoripohjaista rinnakkaisuutta. Stage_event-moduuli



Kuva 5.1: Stage-pelimoottorin komponenttikerrokset

tarjoaa viestinvälitykseen tarvittavia palveluita: viestikanavat, joilla voidaan lähettää viestejä kaikille kiinnostuneille aktoreille, pelimoottorin perusviestitietueet Update, Render, AllDone ja Error, lokipalvelun aktorirajapinnan sekä viestikonteksteja (ks. luku 5.4, Tapahtumakontekstit) ylläpitävät tietorakenteet. Luvussa 2.2 esitellyn pelimoottorien kerrosarkkitehtuurin mukaan viestijärjestelmät ovat yleensä pelimoottorin korkean tason moduuleja, mutta Stage-moottorissa viestinvälityksen apupalvelut ovat kaikkien muiden aktoripohjaisten komponenttien pohjana, koska aktoripohjaisessa toteutuksessa lähes kaikki pelimoottorin osien ja pelioliokomponenttien välinen vuorovaikutus pohjautuu viestinvälitykseen. Stage_scene-moduuli sisältää luokat, joita tarvitaan pelimaailman mallintamiseen: pelialuetta kuvaava Scene, peliolioita ja niiden komponentteja esittävät GameObject ja Component, peliolion sijaintia pelimaailmassa tallentava Component-luokan aliluokka Transform sekä pelialueista kirjaa pitävä abstrakti SceneManager-rajapintaluokka, jonka korkeampaan kerrokseen sisältyvä pelisilmukkaluokka toteuttaa. Pelimoottorikohtaisten moduulien huipulla on kaikki muut moduulit kokonaisuudeksi yhdistävä Stage_core – se sisältää varsinaisen pelisilmukan toteuttavan Gameloop-luokan ja useita luokkia, jotka tarjoavat aktoripohjaiset

rajapinnat yleiskäyttöisen osan grafiikka- ja fysiikkakomponenteille.

Kuvassa 5.1 on lisäksi myös yksi moduuli, joka ei varsinaisesti kuulu moottoriin: Stage_demo sisältää toteutuksen teknologiademolle, jota on tässä tutkielmassa käytetty mittaamaan Stage-pelimoottorin suorituskykyä (ks. luku 6). Varsinaisen demo-ohjelman lisäksi moduuli sisältää demon käyttämät 3D-mallit ja sävytinohjelmat, peliolioon liitettävän komponentin, joka mahdollistaa kameran liikuttamisen näppäimistöllä, sekä tehdasluokan demon käyttämien peliolioiden luomista varten.

5.2 Hienojakoisuus

Stage-pelimoottorissa pelialueet, pelioliot ja niiden sisältämät komponentit on kaikki toteutettu aktoreina, jotka ovat vuorovaikutuksessa yksinomaan viestien välityksellä. Moottorissa on siis käytetty tietopohjaista rinnakkaistamista ja hienojakoisimmaksi rinnakkaisuuden yksiköksi valittiin pelioliokomponentti. Suurimpana syynä tähän oli se, että komponenttipohjaista peliolioiden koostamista käyttävissä pelimoottoreissa pelioliot toimivat pääasiassa tyhjinä ”laatikkoina”, jotka sisältävät varsinaisen toiminnallisuuden ja käyttäytymistavat mallintavia komponentteja, mutta joilla on itse vain hyvin vähän toiminnallisuutta. Toisin sanoen suurin osa moottorissa esiintyvistä vuorovaikutuksista ei ole peliolioiden, vaan niiden sisältämien komponenttien välisiä. Tämä on luontevinta mallintaa aktorijärjestelmässä toteuttamalla peliolioiden lisäksi myös komponentit toistensa kanssa suoraan kommunikoivina aktoreina.

Vaihtoehtoisesti pelioliokomponentit voitaisiin toteuttaa peliolioiden sisäisinä rakenteina, jolloin hienojakoisin rinnakkaisuuden yksikkö olisi peliolio ja kaikki yksittäisen peliolion sisältämät komponentit toimisivat samassa säikeessä. Tämä saattaisi parantaa suorituskykyä ja helpottaa ohjelmointia silloin, kun komponentin laskenta on kevyttä ja käyttää paljon saman peliolion muiden komponenttien tarjoamia palveluita, koska komponentti kykenisi käyttämään muiden saman peliolion komponenttien metodeja suoraan ilman ylimääräisten viestien luomista. Toisaalta tällainen ratkaisu vaikeuttaa viestien käsittelyä, sillä viestit on tällöin kyettävä ohjaamaan peliolion sisällä oikealle komponentille ja siksi jokaisen viestin on sisällettävä myös tiedot viestin lähettäneestä ja sen vastaanottavasta komponentista. Stage-moottorista luotiin myös pelioliota rinnakkaisuusyksikkönä käyttävä versio, mutta kun sen suorituskykyä mitattiin demo-ohjelmalla (ks. luku 6.1, Pelimoottoridemo), se osoittautui alkuperäistä, komponentteja

rinnakkaisuusyksikköinä käyttävää versiota hitaammaksi kaikissa testeissä. Tämä johtui todennäköisesti juuri siitä, että viestien käsittely oli raskaampaa. Pelioliotason hienojakoisuus saattaisi silti olla järkevä ratkaisu, mikäli liika hienojakoisuus osoittautuu vakavaksi suorituskykyongelmaksi, mutta se vaatisi toimiakseen pohjanaan olevaan aktorijärjestelmään jonkinlaisia muutoksia, jotka mahdollistavat viestien lähettämisen tehokkaasti suoraan peliolioiden komponenteille ilman, että saman peliolion eri komponentit voivat joutua samanaikaisesti suoritusvuoroon eri säikeissä.

Kolmas vaihtoehto olisi tehtäväpohjainen rinnakkaistaminen, jossa kokonaiset pelimoottorimoduulit toteutaan yksittäisinä, suurina aktoreina. Tällöin moduulin sisällä laskenta olisi yksisäikeistä ja aktorijärjestelmä toimisi rajapintana, jonka avulla moduulit käyttävät toistensa tarjoamia palveluita. Moduulin laskenta voidaan joko toteuttaa aktorissa, tai aktori voi olla yksinkertainen rajapinta, joka muuttaa aktorijärjestelmän viestit varsinaisen moduulin toteuttamiksi funktiokutsuiksi ja lähettää niiden tuottamat paluuarvot takaisin aktoriviesteinä. Tämä lähestymistapa tarjoaisi luvussa 4.4 listatuista aktorimallin eduista tapahtumajärjestelmän helpon toteuttamisen sekä yleisten rinnakkaisuusongelmien välttämisen, mutta samalla menetettäisiin aktori- ja peliolioabstraktioiden yhteensopivuus sekä aktorijärjestelmien hyvä skaalautuvuus. Täysin tehtäväpohjainen aktoririnnakkaistaminen olisi kolmesta vaihtoehdosta todennäköisesti helpoin ottaa käyttöön nykyisissä kaupallisissa pelimoottoreissa, etenkin, mikäli ne käyttävät paljon kolmansien osapuolien toimittamia ulkoisia apumoduuleja ja -kirjastoja, joiden sisäiseen toteutukseen ei haluta tai voida tehdä muutoksia. Koska samalla kuitenkin menetetään joitakin aktoripohjaisuuden mahdollisia etuja, etenkin tulevaisuuden erittäin moniytimisillä laitteistoalustoilla hyödyllinen skaalautuvuus, se ei ole tutkimuskohteena yhtä kiinnostava kuin hienojakoisemmat mallit.

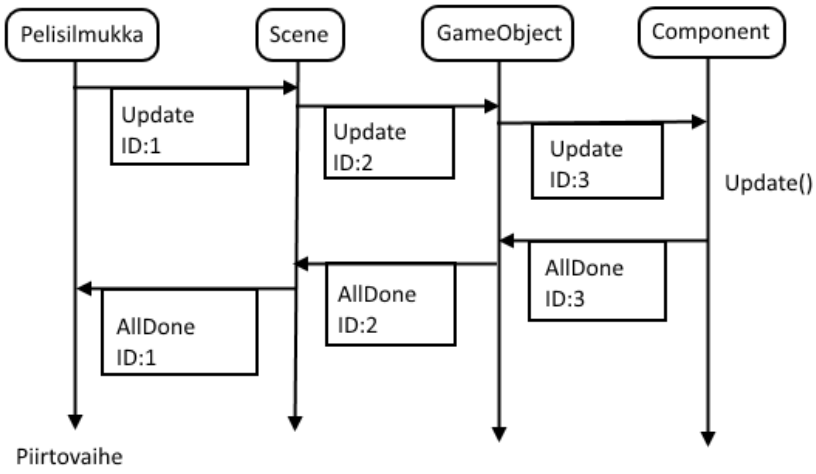
Käytännössä myös peliolio- tai komponenttitason hienojakoisuutta käyttävät aktoripohjaiset pelimoottorit voivat toteuttaa ja todennäköisesti myös joutuvat toteuttamaan osan toiminnoistaan moduulitason tehtäväpohjaisen rinnakkaistamisen kautta, sillä ulkoisia apumoduuleja ja -kirjastoja, joita ei ole suunniteltu säieturvallisiksi, on käytettävä aktorirajapinnan kautta. Esimerkiksi Stage-moottorin grafiikkamoduuli on toteutettu näin: koska Stage-moottorin pohjana toimiva GLFW-kirjasto sallii grafiikan piirtämisen vain ohjelman pääsäikeestä, on 3D-mallien piirtopyynnöt lähetettävä rajapintana toimivalle GraphicsControlActor-aktorille, joka välittää ne säieturvallisesti varsinaiselle grafiikka-

moottorille.

5.3 Moottorin toimintaperiaate

Useimpien Stage-moottorin viestien yliluokkana toimii Event-tietue, joka on määritelty Stage_event-moduulin CoreEvents.h-tiedostossa. Event sisältää ainoastaan kyseisen viestin kontekstin yksilöivän 64-bittisen tunnuksen, joka on muodostettu alkuperäisen kontekstin luojan Theron-osoitteesta sekä 32-bittisestä aktorikohtaisesta viestitunnuksesta. Konteksteja käytetään vastausviestien käsittelyyn ja viestiketjuihin liittyvien muuttujien ylläpitämiseen: koska aktori saattaa käsitellä useaa samankaltaisten viestien ketjua samanaikaisesti, se tarvitsee viestitunnuksia tietääkseen, mihin ketjuun mikäkin viesti kuuluu (ks. luku 5.4, Tapahtumakontekstit). CoreEvents.h sisältää myös yleisimmät moottorissa käytetyt viestit (Update, Render, AllDone ja Error), jotka on kaikki toteutettu Event-tietueen aliluokkina. Lisäksi eri komponenteilla on omia viestitietueitaan, joiden avulla käytetään kyseisen komponentin tarjoamia palveluita; esimerkiksi peliolion sijaintia 3D-maailmassa kuvaava Transform-komponentti määrittelee viestejä kuten SetMatrix (asettaa olion sijaintimatriisille uuden arvon), GetPosition (pyytää olion sijaintia ilman asento- ja skaalaustietoja) ja Position (vastausviesti edelliseen).

Stage-pelimoottori otetaan käyttöön luomalla ensin Gameloop-olio, pelialue sekä peliolioita komponentteineen ja käynnistämällä sitten pelisilmukka Gameloop-olion start-metodilla. Stagen pelisilmukka jakautuu kolmeen vaiheeseen: päivitys, piirto ja ylläpito. Päivitysvaiheessa pelisilmukka lähettää aktiivista pelialuetta esittävälle aktorille Update-viestin, joka sisältää kontekstitunnuksensa lisäksi edellisestä ruudunpäivityksestä kuluneen ajan. Pelialue lähettää viestin edelleen sisältämilleen peliolioille, jotka lähettävät sen komponenteilleen. Saatuaan Update-viestin komponentti tekee tilansa päivittämiseen vaaditut toiminnot, jotka voivat sisältää viestivälitteistä vuorovaikutusta muiden aktorien kanssa, ja lähettää lopuksi isäntäpelioliolleen vastausviestin: joko AllDone tai, mikäli laskennassa tapahtui virhe, Error. Kun peliolio on saanut kaikilta komponenteiltaan vastauksen Update-viestiin, se lähettää itse AllDone-viestin sen omistavalle pelialueelle. Kun kaikki pelioliot ovat vastanneet aktiivisen pelialueen Update-viestiin, pelialue lähettää pelisilmukalle AllDone-viestin, jolloin pelisilmukka siirtyy seuraavaan vaiheeseen. Päivitysvaiheessa lähetettäviä viestejä on havainnollistettu kuvassa 5.2.



Kuva 5.2: Stage-moottorin päivitysvaihe. ID-attribuutit esittävät viestien kontekstitunnuksia

Päivitysvaiheen jälkeen suoritetaan piirtovaihe, joka toimii lähes samalla tavoin: pelisilmukka lähettää aktiiviselle pelialueelle Render-viesti, joka lähetetään edelleen peliolioille ja niiden komponenteille, jotka vastaavat AllDone-viesteillä laskennan päätyttyä. Koska Stagen pohjana toimiva GLFW-kirjasto sallii OpenGL-komentojen suorittamisen vain pääsäikeestä, pelimaailman sisältämiä 3D-malleja ei vielä piirretä tässä vaiheessa, vaan ne lähetetään pelisilmukkaolion omistamalle GraphicsControlActor-oliolle, joka välittää ne säieturvallisesti varsinaisen grafiikkamoottorin toteuttavalle GraphicsController-oliolle, missä niistä muodostetaan lista. Kun pelisilmukka vastaanottaa laskennan päättymistä ilmaisevan AllDone-viestin, GraphicsController käy läpi 3D-mallilistan ja piirtää mallit ruudulle yksitellen.

Lopuksi vuorossa on ylläpitovaihe, jossa tarkistetaan, onko käyttäjä pyytänyt pelin suorituksen lopettamista ja suoritetaan toimintoja, joita ei säieturvallisuuden vuoksi voida toteuttaa päivitys- ja piirtovaiheissa. Esimerkiksi käyttäjän syötteiden lukeminen on GLFW:n rajoitusten vuoksi mahdollista vain pääsäikeessä. Kun pelisilmukka lukee syötteet ylläpitovaiheessa, ne voidaan tallentaa suojaamatta jaettuun muistiin, koska komponenttiaktorit eivät tee laskentaa – jos syötteet haettaisiin päivitysvaiheessa, komponenttiaktorit saattaisivat samanaikaisesti pyrkiä lukemaan niitä omasta säikeestään, jolloin rinnakkaisuusongelmien välttämiseksi ne pitäisi joko suojata lukoilla tai piilottaa aktorirajapinnan taakse. Myös peliolioiden ja niiden komponenttien tuhoaminen on suoritettava ylläpitovaiheessa, sillä muulloin tuhottavalla peliolio- tai komponenttiaktorilla saattaa yhä olla käsittelyä odottavia viestejä, mikä voi helposti aiheuttaa virhetilanteita.

Fysiikkamoottori

Stage-pelimoottori sisältää kevyen ja yksinkertaisen fysiikkamoottorin, jossa peliolioihin voidaan liittää joko pallon tai laatikon muotoisia törmäyshahmoja, jotka voidaan asettaa liikkumaan pelimaailmassa ja kimpoamaan toisistaan. Koska sekä törmäyshahmot että olioita liikuttavat fysiikkalaskelmat ovat fysiikkamallinnuksen minimalistisuuden vuoksi hyvin yksinkertaisia, valtaosa fysiikkamoottorin ohjelmakoodista liittyy törmäystentunnistusjärjestelmään.

Törmäysten tunnistaminen on periaatteessa yksinkertaista: liikutetaan kaikkia pelimaailman olioita fysiikkamoottorin sääntöjen mukaisesti ja verrataan sitten jokaisen fysiikkaolion törmäyshahmon rajaamaa tilaa 3D-maailmassa jokaisen muun pelimaailman fysiikkaolion törmäyshahmon rajaamaan tilaan, jolloin saadaan selville, törmäsivätkö kyseiset pelioliot [MiI07, s. 232]. Jos fysiikkaolioita on paljon, tämä on selvästikin tehotonta, sillä törmäystestien määrä kasvaa fysiikkaolioiden määrän myötä eksponentiaalisesti [MiI07, s.232]. Tämän vuoksi useimpien pelimoottoreiden törmäystenhallinta toteutetaan tallentamalla kaikki pelimaailman törmäyshahmot erityiseen törmäysten tunnistusta varten optimoituun tietorakenteeseen, jota kutsutaan törmäys- tai fysiikkamaailmaksi [GrJ09, s. 605-606].

Törmäysmaailma voidaan toteuttaa monien eri tietorakenteiden avulla, mutta useimmat niistä pohjautuvat erilaisiin puurakenteisiin, kuten hierarkkisiin rajaushahmoihin (hierarchical bounding volumes) [MiI07, s.233-236], BSP-puihin (binary space partitioning, binäärinen avaruudenjako) [MiI07, s. 251-255] tai 8-puihin (octree) [MiI07, s. 255-258]. Käytetystä tietorakenteesta riippumatta tekniikan tarkoituksena on jakaa pelimaailma alueisiin, jotka sisältävät vain pienen osan fysiikkaolioista – tällöin törmäyksiä etsittäessä voidaan jättää huomiotta ne oliot, jotka eivät mitenkään voi törmätä, koska ne sijaitsevat eri osissa pelimaailmaa [MiI07, s. 232-233]. Esimerkiksi 8-puun juurisolmu kuvastaa koko pelimaailmaa ja jokainen sen kahdeksasta lapsisolmusta esittää yhtä pelimaailman kahdeksasosaa. Näiden solmujen lapsisolmut jakavat vanhempansa osan pelimaailmasta edelleen kahdeksaan osaan ja lopulta puun lehtisolmut sisältävät esittämässään pelimaailman osassa sijaitsevien peliolioiden törmäyshahmot [MiI07, s.255-258]. Tämä mahdollistaa törmäysten tunnistamisen tehokkaasti: jokaisen fysiikkaolion kohdalla tarvitsee tarkistaa vain, törmääkö se muihin saman lehtisolmun törmäyshahmoihin, koska eri lehtisolmujen fysiikkaoliot sijaitsevat eri osissa pelimaailmaa eivätkä

siksi voi mitenkään törmätä.

Stage-pelimoottorin törmäystentunnistus ei hyödynnä puurakennetta, vaan ”törmäysmaailmana” käytetään tapahtumakanavaa (ks. luku 5.4, Tapahtumakanavat aktoriviestinnän laajennoksena), jonka viestien vastaanottajaksi kaikki fysiikkakomponentit rekisteröityvät. Tämä ei missään nimessä ole tehokasta, mutta järjestelmän toimintaperiaate on silti sama kuin muissakin törmäyksentunnistusjärjestelmissä: kun fysiikkaoliota on liikutettu fysiikkamoottorin sääntöjen mukaan, tarkistetaan törmääkö se mihinkään muuhun pelimaailman olioon. Stage-moottorissa tämä on toteutettu siten, että päivitysvaiheen osana peliolion fysiikkakomponentti lähettää tapahtumakanavan kautta törmäyshahmonsa tiedot kaikille muille fysiikkaolioille, jotka tarkistavat törmäyksen ja lähettävät takaisin tarvittavat tiedot törmäyksen käsittelyyn, mikäli törmäys havaitaan. Syitä juuri tämän toteutustavan valitsemiseen oli kolme:

- Tapahtumakanavapohjainen törmäystentunnistusjärjestelmä on huomattavasti yksinkertaisempi toteuttaa, ja vaikka se on muita ratkaisuja tehottomampi, tutkimuksen tavoitteena ei ollut toteuttaa valmista, tehokasta pelimoottoria vaan verrata pelimoottorin aktoripohjaista rinnakkaistamista perinteisiin rinnakkaistamistekniikoihin.
- Eräs aktoripohjaisen toteutuksen valitsemisen syistä oli tapahtumajärjestelmän toteuttamisen ja käyttämisen helppous. Toteuttamalla fysiikkamoottori tapahtumakanavien avulla kyettiin arvioimaan tämän toteutumista käytännössä.
- Tapahtumapohjaisen törmäystentunnistukseen vaadittavan suuren viestimäärän avulla kyettiin testaamaan tapahtumajärjestelmän ja ylipäätään aktoripohjaisen pelimoottoritoteutuksen skaalautuvuutta erittäin suurille määrille viestejä.

Jos aktorien pohjalta ryhdytään kehittämään tehokasta pelimoottoria, tapahtumakanavapohjaisen törmäystentunnistuksen sijaan tulisi käyttää esimerkiksi 8-puita. Tämä on kuitenkin yksinkertaista aktoripohjaisessa toteutuksessa, sillä aktorit soveltuvat hyvin hierarkkisten rakenteiden kuten puiden mallintamiseen. Tällöin törmäyksentunnistusjärjestelmää voidaan käyttää lähes samalla tavalla kuin tapahtumakanavia: kun peliin luodaan törmäyksentunnistusta käyttävä pelioliokomponentti, se lähettää rekisteröintiviestin törmäysmaailmapuun juuriaktorille. Tämä lähettää rekisteröintiviestin fysiikkaolion sijainnin perusteella edelleen yhdelle lapsisolmuistaan, joka lähettää sen

yhdelle omista lapsistaan ja niin edelleen, kunnes viesti lopulta saapuu puun lehtisolmuun. Lehtisolmuaktori lisää fysiikkaolion osoitteen listaan, joka sisältää kyseisen lehtisolmun mallintaman pelimaailman osan kaikki fysiikkaoliot. Jokaisen ruudunpäivityksen aikana fysiikkaoliot päivittävät sijaintinsa ja lähettävät sen niitä hallinnoiville törmäysjärjestelmän lehtisolmuaktoreille, jotka välittävät ne edelleen muille saman lehtisolmun fysiikkaolioille törmäyksentunnistusta varten.

Puupohjaisen törmäyksentunnistusjärjestelmän toteutus aktoripohjaisessa pelimoottorissa itse asiassa helpottuu, mikäli fysiikkamoottorin halutaan deterministisyyden vuoksi toimivan tietyllä määrätyllä ruudunpäivitystaajuudella, muun pelimoottorin ruudunpäivitysnopeudesta riippumatta. Fysiikkamoottorin päivitysviestit voidaan nimittäin lähettää fysiikkaolioille törmäysmaailman kautta siten, että jokainen lehtisolmuaktori välittää päivitysviestin omistamilleen fysiikkaolioille. Tällöin fysiikkakomponentin ei itse tarvitse pitää kirjaa sen omistavasta lehtisolmusta, vaan se voi lähettää tiedon muuttuneesta sijainnistaan vastauksena päivitysviestin lähettäjälle.

Jos moottorissa halutaan käyttää täysin ulkoista fysiikkamoottoria, joka toteuttaa itse oman törmäysmaailmansa ja simulaationsa, käyttöönotto riippuu fysiikkamoottorin tarjoamista rajapinnoista. Yksinkertaisin keino on todennäköisesti käyttää luvussa 5.2 kuvattua moduulitason rinnakkaistamista, jossa luodaan yksi, globaali aktori fysiikkamoottorin ja peliolioiden väliseksi rajapinnaksi, mutta tällöin rajapinta-aktori saattaa helposti muodostua pullonkaulaksi. Vaihtoehtoisesti jokaiselle pelioliolle voidaan antaa oma fysiikkakomponentti, joka käyttää fysiikkamoottorin tarjoamaa rajapintaa itsenäisesti. Tällöin on kuitenkin varmistuttava fysiikkamoottorin rinnakkaisen käytön turvallisuudesta – mikäli fysiikkamoottori ei itse takaa kutsujensa säieturvallisuutta, saatetaan fysiikkakomponenttiaktoreissa joutua käyttämään esimerkiksi lukkoja tai semaforeja.

5.4 Toteutuksen haasteet ja ratkaisut

Tässä luvussa tarkastellaan Stage-pelimoottorin toteutuksessa ilmenneitä ongelmia sekä niihin löydettyjä ratkaisuja.

Tilan synkronointi

Aktorijärjestelmät ovat pohjimmiltaan asynkronisia: siinä missä useimmissa perinteisissä rinnakkaisuusratkaisuissa voidaan jäädä odottamaan laskennan loppumista

esimerkiksi säieolion join-metodilla, aktorit käsittelevät viestejä itsenäisesti ja aktorilaskennan päättymisen saa selville vain, jos aktori päättää itse ilmoittaa siitä. Pelimoottorissa tila on kuitenkin synkronoitava joka ruudunpäivityksen lopuksi: seuraavan ruudun laskemista ei voida aloittaa, ennen kuin edellinen on laskettu loppuun. Tämän vuoksi Stage-pelimoottori asettaa pelin toteuttajalle luvussa 4.3 esitettyjen aktoriohjelmoinnin rajoitusten lisäksi yhden uuden rajoituksen:

- Kaikkien pelimoottorin viestien käsittelyn täytyy päättyä laskennan päättymisestä kertovan vastausviestin lähettämiseen alkuperäisen viestin lähettäjälle.

Pelin toteuttajaa ei voida mitenkään pakottaa noudattamaan tätä sääntöä (kuten ei luvussa 4.3 mainittuja hyvän aktoriohjelmointityylin sääntöjäkään), mutta säännön noudattaminen varmistaa, että edellisen ruudunpäivityksen laskenta saadaan loppuun ennen seuraavan aloittamista. Sääntö varmistaa myös, että peliolioiden komponentit suorittavat laskentaa vain pelisilmukan päivitys- ja piirtovaiheissa. Säännön vuoksi Stage-moottori toimii tiukasti lukittu askel -tilassa: kaikki nykyisen pelisilmukan vaiheen laskenta on suoritettava loppuun ennen seuraavan vaiheen aloittamista.

Vastausviestin tyyppi riippuu alkuperäisestä viestistä. Jos viestinkäsittelijän laskennalla on jokin paluuarvo (esimerkiksi Transform-luokan GetPosition-viesti pyytää vastaanottajaa lähettämään sijaintiaan 3D-maailmassa kuvaavan vektorin), vastausviestiksi riittää laskennan tuloksen sisältävä viesti. Jos laskenta ei tuota erityistä paluuarvoa, tulee viestinkäsittelijän lähettää takaisin AllDone-tietue tai, mikäli käsittelyssä tapahtuu virhe, Error-tietue.

Vastausviestit voisi mahdollisesti jättää lähettämättä, jos pelin ruudunpäivitystaajuus on vakio, eikä tilanpäivityksen laskenta-aika koskaan ylitä ruudunpäivitykselle varattua aikaa. Tällöin ruudunpäivitys on aina laskettu loppuun ennen seuraavan aloittamista, eikä pelisilmukalle siksi tarvitse tiedottaa laskennan päättymisestä. Vaarana on kuitenkin se, että jos ruudunpäivitys kuitenkin kuluttaa enemmän kuin sille varatun ajan, osa laskennasta tapahtuu käytännössä seuraavassa ruudunpäivityksessä, jolloin pelaajalle näkyvä pelitilanne voi hetkellisesti olla ristiriitainen, ja mikäli laskenta jatkuu raskaana, peli hidastuu hidastumistaan ruudunpäivityksistä yli vuotavan työn määrän kasvaessa. Tämän vuoksi tämä lähestymistapa toimisi käytännössä vain hyvin kevyillä peleillä tai pelikonsoleilla, joilla on aina käytettävissään sama laskentateho – kotitietokoneiden ja mobiililaitteiden tapauksessa laskentateho vaihtelee alustan mukaan, joten ruudunpäivi-

tykseen käytössä olevaa aikaa on hankala ennustaa.

Tapahtumakontekstit

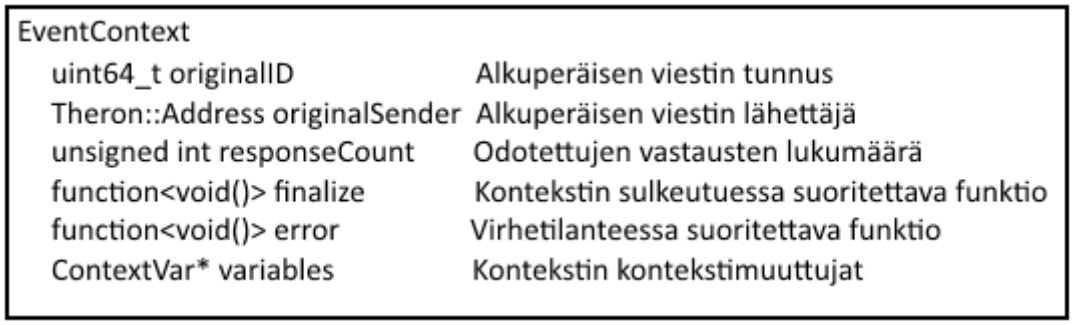
Aktoreita käytetään usein erityisesti liukuhihnatyylisissä sovelluksissa, joissa aktorit prosessoivat saamiaan syötteitä ja lähettävät sitten tulokset muualle – laskennan suoritukseen ja lopputulokseen vaikuttavat tällöin vain syötteen sisältö ja aktorin oma sisäinen tila. Pelimoottoreissa laskenta vaatii kuitenkin usein eri peliolioiden ja niiden komponenttien välistä vuorovaikutusta, sillä jokainen komponentti tuntee itse vain osan peliolion tilasta. Tällöin kaikissa tapauksissa alkuperäisen viestin pyytämä laskenta ei päätykään viestinkäsittelijän suorituksen loppuessa: esimerkiksi Stage-moottorin peliolioiden fysiikkakomponentin (ks. luku 5.3, Fysiikkamoottori) on joka ruudunpäivityksen aikana Update-viestin vastaanotettuaan päivitettävä sijaintinsa, kysyttävä muilta fysiikkakomponenteilta, aiheuttaako uusi sijainti törmäyksen niiden kanssa, ja törmäysten käsittelyn jälkeen vastattava alkuperäiseen Update-viestiin AllDone-viestillä. Laskennan päättymistä ilmaiseva AllDone-viesti voidaan siis lähettää vasta, kun kaikki muut fysiikkakomponentit ovat vastanneet, ja koska aktorit eivät saa jäädä odottamaan toisiaan, alkuperäisen Update-viestin käsittelijän suorituksen on päätyttävä, ennen kuin muiden fysiikkakomponenttien vastaukset voidaan käsitellä. AllDone-vastausviesti on siis lähetettävä jostakin muualta kuin Update-viestin käsittelijästä – tämä on ongelma, koska tällöin alkuperäisen viestinkäsittelijän paikallisiin muuttujiin, kuten Update-viestin lähettäjän osoitteeseen (eli vastausviestin osoitteeseen), ei enää päästä käsiksi.

Useimmissa aktorisovelluksissa tällaiset tilanteet ratkaistaan tallentamalla viestiketjussa myöhemmin tarvittavien muuttujien arvot aktorin sisäisiin kenttiin, mikä saattaisi myös pelimoottorissa toimia hyvin pienillä komponenteilla. Pelimoottoreiden monimutkaisuudesta johtuen poiketen pelioliokomponentit eivät usein kuitenkaan suorita vain yhtä tehtävää, vaan saattavat joutua käsittelemään monia erilaisia rooliinsa liittyviä viestiketjuja, jolloin viestiketjujen hallintaan väliaikaisesti tarvittavien sisäisten kenttien määrä kasvaa helposti. Lisäksi pelioliokomponentit saattavat joutua käsittelemään useita samanlaisia viestiketjuja samanaikaisesti, minkä vuoksi näihin väliaikaisia muuttujia ylläpitäviin kenttiin on voitava tallentaa useita arvoja, jotka on lisäksi kyettävä yhdistämään oikeisiin viestiketjuihin.

Ongelman ratkaisemiseksi on Stage-moottoria varten kehitetty tapahtumakonteksteiksi (EventContext) kutsutut aputietueet sekä niitä hallinnoivat ContextTracker-oliot. Tapah-

tumakontekstit ovat tietorakenteita, joiden avulla aktori kykenee pitämään muistissa muuttujia useiden toisiinsa liittyvien viestien muodostamien viestiketjujen käsittelyn ajan, ja jokaisella pelioliolla ja komponentilla on kyseisen aktorin avoimista konteksteista kirjaa pitävä ContextTracker-olio. Kun yhteen viestiin vastaamiseen vaaditaan useiden viestien käsittelyä, ContextTracker-olion ylläpitämään hajautustauluun luodaan uusi kontekstiolio, joka pitää muistissa muun muassa alkuperäisen viestin kontekstitunnusta ja lähettäjää. Kun alkuperäisen viestin käsittely on saatu valmiiksi ja siihen voidaan lähettää vastaus, vastausviestin tunnus ja osoite voidaan hakea kontekstioliosta, joten vastaus voidaan lähettää mistä tahansa kontekstin omistavan aktorin metodista. Viestitietueiden 64-bittistä kontekstitunnusta käytetään yhdistämään vastausviesti sitä pyytäneen viestin kontekstiin: kun aktori lähettää vastausviestin, vastauksen kontekstitunnuksena käytetään alkuperäisen viestin tunnusta. Esimerkiksi peliolion fysiikkakomponentti saattaa käsitellä samanaikaisesti törmäyksiä useiden muiden fysiikkaolioiden kanssa – tällöin kontekstitunnuksen avulla erotetaan, mitkä viestit liittyvät mihinkin törmäystapahtumaan ja estetään eri törmäyksiin liittyvien tietojen sekoittuminen.

Tapahtumakontekstin rakenne on esitetty kuvassa 5.3. Vastausviestin lähettämistä varten kontekstiin on tallennettava alkuperäisen lähettäjän osoite ja viestin tunnus – ilman ensimmäistä viestiä ei voida ohjata oikealle aktorille ja ilman jälkimmäistä vastaanottaja ei tiedä, mihin sen lähettämään viestiin vastataan. Nämä tiedot tallennetaan tapahtumakontekstin kenttiin originalSender ja originalID. Tapahtumakontekstissa on lisäksi useita apumuuttujia ohjelmoinnin helpottamiseksi. responseCount-muuttujaan tallennetaan kontekstiin odotettujen vastausviestien määrä, mikä on hyödyllistä esimerkiksi peliolion lähettäessä Update-viestejä edelleen komponenteilleen: joka kerta, kun komponentilta saadaan laskennan päättymistä ilmaiseva vastaus, muuttujan arvosta vähennetään yksi,



Kuva 5.3: Tapahtumakontekstin sisältö

jolloin muuttujan saadessa arvon 0 peliolio on saanut vastauksen kaikilta komponenteiltaan ja voi itse ilmoittaa laskennan päättyneen omalle omistajalleen. Täten jokaista komponenttia varten ei tarvitse luoda uutta kontekstia, vaan kaikki vastaukset voidaan käsitellä yhden kontekstin avulla.

Vastausviestien lähettämistä on edelleen virtaviivaistettu finalize-muuttujalla, johon tallennettava funktio suoritetaan automaattisesti responseCount-muuttujan laskiessa nollaan ja oletuksena lähettää alkuperäiselle lähettäjälle laskennan päättymisestä kertovan AllDone-viestin. error-muuttuja toimii samoin, mutta sen sisältämä funktio suoritetaan aktorin vastaanottaessa kontekstiin liittyvän laskennassa tapahtuneesta virheestä kertovan Error-viestin. Näiden lisäksi jokaiseen kontekstiin voidaan liittää mielivaltaisia kontekstimuuttujia, jotka toimivat käytännössä samaan kontekstiin liittyvien viestinkäsittelijöiden jakamina paikallisina muuttujina, joihin päästään käsiksi kontekstin aksessorimetodien kautta.

Yksinkertaisten tapahtumakontekstien toimintaa havainnollistetaan listauksessa 5.1. Peliolioon kameran liittävän CameraComponent-olion on aina moottorin piirtovaiheessa haettava isäntäolionsa sijaintikomponentilta peliolion nykyinen sijainti, jotta pelimaailma voidaan piirtää oikeasta kuvakulmasta. Tämä tapahtuu kamerakomponentin render-metodissa. Komponentti pyytää ensin ContextTracker-olioltaan käyttämättömän kontekstitunnuksen ja luo sitten sen perusteella uuden tapahtumakontekstin, jolle annetaan parametreiksi myös render-metodin käynnistäneen piirtopyynnön kontekstitunnus ja lähettäjä. Lopuksi render-metodi lähettää sijaintikomponentille viestin, jossa pyydetään sitä ilmoittamaan peliolion nykyinen sijaintimatriisi.

Kun sijaintiolion vastaus saapuu, se käynnistää completeRender-viestinkäsittelijän. Metodi asettaa kamerakomponentin kameraoliolle uuden näkymämatriisin vastauksen sisältämän sijainnin perusteella ja vähentää sitten viestin osoittaman kontekstin odotettujen vastausten määrää yhdellä. Koska vastaanotetun viestin tunnus on sama kuin render-metodissa luodun kontekstin, juuri tämän kontekstin vastausmäärää vähennetään, ja koska kyseisen kontekstin odotettujen vastausten määrä oli oletusarvo 1, se laskee nollaan. Tällöin ContextTracker suorittaa automaattisesti kontekstin finalize-funktion, joka oli määritelty kontekstin luoneessa ContextTracker-olion addContext-metodissa. Oletus-finalize hakee kontekstista alkuperäisen Render-viestin lähettäjän ja kontekstitunnuksen ja lähettää sitten niiden perusteella AllDone-viestin, joka ilmoittaa Render-

```
//CameraComponent.cpp
//---Kontekstiyksikkö alkaa--
void CameraComponent::render(const Render& msg, Theron::Address sender){
	/*...*/
	uint64_t id = tracker.getNextID();
	tracker.addContext(msg.id, id, sender);
	//Haetaan sijaintikomponentilta kameran sijainti tämän ruudunpäivityksen
	//aikana
	Send(Transform::GetMatrix(id), transform);
}

void CameraComponent::completeRender(const Transform::Matrix& msg,
Theron::Address sender){
	/*...*/
	//Asetetaan kameralle uusi sijainti
	cam.setViewMatrix(msg.matrix);
	tracker.decrement(msg.id);
}
//---Kontekstiyksikkö päättyy--

//ContextTracker.cpp

EventContext& ContextTracker::addContext(uint64_t oldID, uint64_t newID,
Theron::Address originalSender, int responseCount = 1){
	//Luodaan uusi konteksti
	pending[newID] = EventContext(oldID, originalSender, responseCount);
	EventContext& context = pending[newID];
	//Asetetaan oletuskäsittelijä kontekstin loppuunsuorittamiselle:
	//palautetaan AllDone alkuperäiselle lähettäjälle
	context.finalize = [this, &context](){
		this->fw.Send(AllDone(context.getOriginalID()), this->owner,
context.getOriginalSender());
	};
	context.error = context.finalize;
	return context;
}

void ContextTracker::decrement(uint64_t id){
	EventContext& context = pending[id];
	context.responseCount--;
	//Suljetaan konteksti, kun vastauksia ei odoteta enempää
	if (context.responseCount < 1){
		context.finalize();
		pending.erase(id);
	}
}
```

Listaus 5.1: Esimerkki tapahtumakonteksteja hyödyntävästä aktorista

viestin lähettäjälle kamerakomponentin saaneen piirtovaiheen laskentansa onnistuneesti päätökseen.

Yhden viestiketjun käsittelyyn käytettävien metodien muodostamat kontekstiyksiköt muistuttavat periaatteeltaan vuorottaisrutiineja (coroutines), sillä molemmissa suoritetaan ohjelmaa jonkin aikaa, pysäytetään suoritus ja jatketaan myöhemmin samasta ohjelman kohdasta. Tapahtumakontekstit olisikin mahdollista toteuttaa vaihtoehtoisesti

vuorottaisrutiineina, mikä voisi parantaa ohjelmakoodin luettavuutta, koska yhden viestiketjun käsittelylogiikkaa ei tarvitsisi pilkkoa useiksi viestinkäsittelijämetodeiksi. C++-kielessä ei kuitenkaan vielä ole sisäänrakennettua tukea vuorottaisrutiineille, ja tapahtumakontekstit ovat joustavampia ohjelman rakenteen suhteen, sillä erillisillä viestinkäsittelijämetodeilla voidaan toteuttaa helpommin esimerkiksi eri tyyppisten vastausviestien käsitteleminen viestiketjun eri vaiheissa sekä useiden kerralla lähetettyjen viestien vastausten käsittely saapumisjärjestyksestä riippumatta. Joka tapauksessa myös vuorottaisrutiineilla toteutetut kontekstit vaatisivat toimiakseen kontekstitunnuksia ja konteksteista kirjaa pitäviä tietorakenteita, sillä yksittäisellä aktorilla saattaa olla samanaikaisesti suoritettavana useita samankaltaisia viestiketjuja eri suoritusvaiheissa.

Tapahtumakanavat aktoriviestinnän laajennoksena

Monissa pelimoottoreissa hyödynnetään tapahtumajärjestelmiä, joiden avulla pelioliot tai niiden komponentit voivat rekisteröityä ottamaan vastaan ilmoituksia tietyistä tapahtumista, joihin ne haluavat reagoida. Esimerkiksi jos hiiviskelypelistä tekoälyviholliseen halutaan reagoivan pelaajan aiheuttamiin ääniin, se voidaan rekisteröidä ottamaan vastaan viestejä, joita luodaan aina, kun pelaajan hahmo pitää meteliä. Aktorijärjestelmä tarjoaa tapahtumajärjestelmälle hyvän pohjan, sillä se pohjautuu itsekin viestinvälitykseen ja tarjoaa siten valmiin, rinnakkaisen kehyksen viestien lähettämistä, vastaanottamista ja käsittelyä varten. Aktorijärjestelmä vaatii kuitenkin lähettäjää tietämään vastaanottajan osoitteen, kun taas pelimoottorien tapahtumajärjestelmissä mikä tahansa olio voi lähettää järjestelmän kautta viestin kaikille siitä kiinnostuneille ilman, että sen täytyy tuntea vastaanottajia. Tämän toiminnallisuuden toteuttamiseksi Stage-moottorissa laajennettiin aktorijärjestelmän viestinvälitystä tapahtumakanavilla, jotka pitävät kirjaa viestityyppien rekisteröityneistä vastaanottajista ja ohjaavat niille halutut viestit.

Yksinkertaisin tapa toteuttaa tapahtumakanava on luoda sitä mallintava aktori, joka välittää kaikki tietyntyyppiset viestit kaikille niistä kiinnostuneille vastaanottajille. Tietyntyyppisistä tapahtumista kiinnostunut aktori lähettää ensin rekisteröitymisviestin kyseisiä tapahtumia käsittelevälle tapahtumakanava-aktorille, joka lisää sen vastaanottajalistaansa. Kun jokin peliolio aiheuttaa tapahtuman, se lähettää siitä ilmoituksen kyseisen tyyppisiä tapahtumia käsittelevälle kanavalle, joka lähettää tapahtumailmoituksesta kopiot edelleen kaikille niille aktoreille, jotka on rekisteröity sen

vastaanottajalistaan. Suurilla viestimäärillä tämä ratkaisu on kuitenkin tehoton: kun sitä yritettiin soveltaa Stage-moottorin fysiikkamoottoriin (ks. luku 5.3, Fysiikkamoottori), jossa jokainen fysiikkaolio lähettää ilmoituksen liikkeistään kaikille muille jokaisen ruudunpäivityksen aikana törmäystentunnistusta varten, moottorin suorituskyky heikkeni fysiikkaolioiden määrän kasvaessa nopeasti. Tämä johtui siitä, että samalla fysiikkamoottorin viestien määrä kasvoi eksponentiaalisesti, ja koska tapahtumakanava joutui itse välittämään ne kaikki vastaanottajilleen, siitä muodostui pullonkaula.

Pullonkaula saatiin poistettua antamalla fysiikkakomponenteille suora lukuoikeus tapahtumakanavan vastaanottajalistaan: kun fysiikkaoliot hakivat toistensa osoitteet listasta ja lähettivät tapahtumailmoituksensa toisilleen suoraan omasta säikeestään, tapahtumakanava-aktorin ei tarvinnut tehdä mitään ja käsiteltävät fysiikkamoottorin viestit jakautuivat fysiikkakomponentteja suorittavien säikeiden välille tasapuolisesti. Tämä aiheutti kuitenkin toisen ongelman, sillä nyt fysiikkakomponentit pääsivät lukemaan suoraan tapahtumakanavan paikallisessa muistissa sijaitsevaa listaoliota, mikä olisi voinut aiheuttaa rinnakkaisuusongelmia, jos vastaanottajalistan sisältöä olisi muokattu ohjelman suorituksen aikana. Ratkaisu tähän oli kuitenkin yksinkertainen pelisilmukan rakenteen ansiosta: vastaanottajalistan päivitys eli tapahtumakanavaan rekisteröityneiden vastaanottajien lisäys ja poisto voidaan suorittaa turvallisesti silmukan ylläpitovaiheessa, koska peliolioiden komponentit saavat suorittaa laskentaa vain pelisilmukan päivitys- ja piirtovaiheissa, eivätkä siten voi lukea listan sisältöä näiden vaiheiden ulkopuolella.

Stage on aktoripohjainen pelimoottori, jonka avulla tutkitaan, miten hyvin aktoripohjainen rinnakkaisuus soveltuu pelimoottorien toteuttamisen pohjaksi. Sen kerrosarkkitehtuuri muistuttaa paljon muita pelimoottoreita, mutta sen pelialueet, peliolio t ja niiden komponentit on toteutettu aktoreina, jotka ovat toistensa kanssa vuorovaikutuksessa lähinnä viestinvälityksen avulla. Pelitila on pelimoottoreissa synkronoitava joka ruudunpäivityksen aikana, ja Stage-moottorissa tämä tapahtuu hyödyntämällä tapahtumakonteksteja. Ne pitävät kirjaa tapahtumien käsittelyn etenemisestä ja tapahtumiin liittyvistä muuttujista useiden eri viestinkäsittelijöiden suorituksen aikana, mutta vaativat toimiakseen, että pelin toteuttaja pitää huolen siitä, että jokaisen tapahtuman käsittely päättyy vastausviestin lähettämiseen tapahtuman aiheuttaneelle aktorille.

6 Toteutuksen arviointi

Jotta aktoripohjaisen pelimoottoriarkkitehtuurin mielekkyyttä voidaan arvioida, sitä on verrattava muihin rinnakkaistamisratkaisuihin. Markkinoilla olevat valmiit pelimoottoritoteutukset sisältävät kuitenkin paljon enemmän ja monimutkaisempia komponentteja, mikä tekee niiden laskennasta raskaampaa, ja toisaalta monet niistä hyödyntävät matalan tason optimointeja kuten omia muistinhallintajärjestelmiä, jotka lisäävät niiden tehokkuutta. Tämän vuoksi niiden vertaaminen Stage-moottorin kanssa ei ole mielekästä, sillä olisi hankala arvioida, miten suuri vaikutus aktoripohjaisuudella olisi loppujen lopuksi tuloksiin. Vertailu on huomattavasti järkevämpi, kun vertailumoottorit sisältävät täsmälleen samat ominaisuudet ja perustuvat samoihin kirjastoihin. Tässä luvussa esitellään lyhyesti vertailua varten kehitetyt pelimoottorit Stage Control ja Stage 11 sekä tutkitaan Stage-moottorin tehokkuutta ja sen avulla luotujen ohjelmien toteuttamisen työläyttä näihin verrattuna.

6.1 Vertailumoottorit ja koeasettelu

Aktoripohjaisen lähestymistavan mielekkyyden arviointia varten kehitettiin kaksi vertailumoottoria: Stage Control on yksisäikeinen, rinnakkaistamaton toteutus, kun taas Stage 11 on rinnakkaistettu C++11:n säikeiden avulla. Pelimoottoritoteutusten arvioimiseksi kaikilla moottoreilla toteutettiin sama teknologiademo.

Stage Control

Stage Control -moottorin yleiskäyttöinen osa on täysin identtinen Stage-moottorin kanssa (ks. kuva 5.1, luku 5.1), ja sen moottorikohtainen osa sisältää samat moduulit toteutettuina yksisäikeisinä. Kuten Stage, Stage Control käyttää komponenttipohjaista peliolioiden koostamista, jossa peliolioiden toiminnallisuus riippuu niiden omistamista komponenttiolioista, mutta yksisäikeisenä se ei tarvitse tapahtumakonteksteja tai muita Stage-moottorin käyttämiä viestinvälityksen apumekanismeja, sillä komponenttiolioiden välinen vuorovaikutus tapahtuu pääasiallisesti suorien metodikutsujen kautta. Tehokkuusvertailun pitämiseksi järkevänä Stage Control -moottorin törmäystentunnistusjärjestelmä on toteutettu tapahtumajärjestelmän avulla, aivan kuten Stage-moottorissakin (ks. luku 5.2, Fysiikkamoottori). Stage Control -moottorin ohjelmakoodi on liitteessä 3.

Stage 11

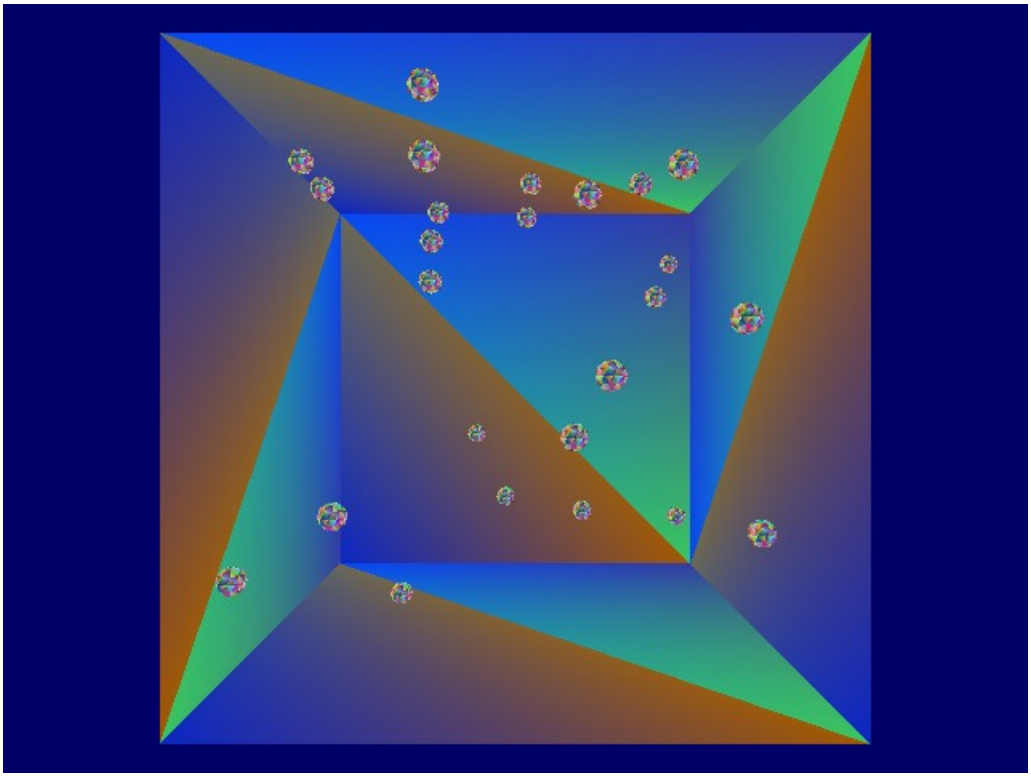
Myös Stage 11 -moottorin yleiskäyttöinen osa on identtinen Stage-moottorin kanssa, mutta sen moottorikohtainen osa on toteutettu C++11:n säiekirjaston avulla. Stage-moottorin sisältämien moduulien lisäksi sen pelimoottorikohtainen osa sisältää yleiskäyttöisen osan ja muiden pelimoottorikohtaisten moduulien välissä toimivan Stage_11_thread-moduulin, joka toteuttaa moottorin pohjana toimivan säiealtaan.

Stage 11 -moottorissa peliolioiden ja niiden komponenttien Update- ja Render-kutsut sekä tapahtumajärjestelmän kautta lähetettävät viestit paketoidaan tietueiksi, joita kutsutaan töiksi (task). Tehtävät lisätään säiealtaan hallinnoimaan työlistaan, josta lepotilassa olevat työntekijäsäikeet hakevat ja suorittavat ne. Kun työlista on tyhjä ja kaikki työntekijäsäikeet ovat lepotilassa, pääohjelmasäie käynnistää pelisilmukan seuraavan vaiheen. Tehtävien sisällä rinnakkaisuutta hallinnoidaan C++11:n mutex-lukoilla, joiden avulla varmistetaan, ettei samojen muuttujien arvoihin päästä käsiksi useammasta säikeestä yhtä aikaa. Pienin rinnakkaisuuden yksikkö Stage 11-moottorissa on siis yksittäinen päivitys-, piirto- tai tapahtumankäsittelymetodin kutsu. Vaikka ratkaisu perustuu Stage-moottorin rinnakkaistamismalliin, se on käytännössä karkeajakoisempi, sillä Stage 11:n työtehtävien sisällä säikeen ei tarvitse jäädä odottamaan tai antaa suoritusvuoroa muille säikeille, ellei se yritä saada itselleen toisen säikeen varaamaa lukkoa – Stage-moottorissa puolestaan yhtä työtehtävää vastaavan viestiketjun jokainen viesti on käsiteltävä juuri vastaanottaja-aktoria suorittavassa säikeessä.

Kuten Stage- ja Stage Control -moottoreissa, Stage 11 -moottorin törmäystentunnistusjärjestelmä on toteutettu tapahtumajärjestelmän avulla, jotta eri moottoritoteutusten välinen vertailu säilyisi mahdollisimman mielekkäänä. Stage 11 -moottorin ohjelmakoodi on liitteessä 4.

Pelimoottoridemo

Pelimoottoritoteutusten vertailussa käytetyssä demo-ohjelmassa on litteistä pinnoista muodostettu kuution muotoinen laatikko, jonka sisällä on palloja. Laatikon koko ja pallojen sekä käytettävien säikeiden määrä luetaan demon käynnistyessä config.ini-nimisestä tiedostosta. Pallot lähtevät demon alussa liikkumaan satunnaiseen suuntaan satunnaisella nopeudella ja törmätessään toisiinsa tai laatikon seiniin ne kimpoavat Sta-



Kuva 6.1: Kuvankaappaus Stage-pelimoottorin demo-ohjelmasta

ge-moottorin sisältämän yksinkertaisen fysiikkamoottorin sääntöjen mukaisesti. ”Pelaaja” voi liikuttaa demon kameraa edestakaisin ja sivuttain WASD-näppäimillä, minkä lisäksi R- ja F-näppäimet liikuttavat kameraa ylös ja alas. Demon suorituksen voi pysäyttää Escape-näppäimellä, jolloin pelimoottori näyttää viimeisen suorituskerran suorituskykytietoja: ruudunpäivityksiin keskimäärin kuluneen ajan sekä pelisilmukan eri suoritusvaiheisiin (päivitys, piirto ja ylläpito) kuluneen ajan.

Demo-ohjelman avulla verrataan pelimoottoritoteutuksia kahdella tavalla: demon toteutuksessa ilmenneiden haasteiden kautta verrataan, miten hankalaa eri moottoreiden avulla on toteuttaa peli, ja demon suorittamisen aikana kerätyillä suorituskykytiedoilla verrataan, miten tehokkaita ja skaalautuvia toteutukset ovat.

6.2 Ratkaisujen vertailu

Erikoistamisvaikeus

Koska pelimoottori ei ole itsessään hyödyllinen ohjelma, vaan pohja sen päälle toteutettaville peleille, on tärkeää tarkastella sen erikoistamisvaikeutta, eli sitä, miten vaikeaa sen käyttö on ja miten paljon vaivaa pelien toteuttaminen sen avulla vaatii. Stage-moottorin tapauksessa tätä tutkittiin demo-ohjelman toteuttamisen yhdessä: koska sama

demo toteutettiin sekä Stage-moottorin että sen kahden vertailutoteutuksen avulla, kyettiin arvioimaan, miten aktoriohjelmointityyli ja aktoriohjelmoinnille ominaiset haasteet vaikuttivat ohjelman toteuttamisen hankaluuteen yksisäikeiseen ja lukkopohjaiseen ohjelmointiin verrattuna.

Kolmesta moottorista Stage Control on selvästi yksinkertaisin pelin toteuttamisen kannalta: koska moottori on yksisäikeinen, rinnakkaisuutta ei tarvitse ottaa mitenkään huomioon, vaan peli voidaan toteuttaa kuten mikä tahansa C++-ohjelma. Stage 11-moottorin avulla toteutettu peliohjelma muistuttaa paljon Stage Control -moottorin avulla toteutettua, mutta siinä on lisäksi huolehdittava säieturvallisuudesta. Useimmissa tapauksissa tämä tarkoittaa vain, että pelioliokomponentti sisältää mutex-lukon, joka lukitaan komponentin metodien sisällä ennen komponentin sisäisen tilan muuttamista tai lukemista – C++11:n unique_lock-lukon toteutus varmistaa, että lukko vapautetaan jälkeenpäin. Tällöin toteuttajan ei tarvitse varmistaa metodikutsun säieturvallisuutta ennen komponentin palvelujen käyttämistä sen ulkopuolelta. Joissakin tapauksissa rinnakkaisuus edellyttää kuitenkin monimutkaisempia toimia: esimerkiksi tapahtumakanavat käyttävät luku-kirjoituslukkoja, jotka on manuaalisesti lukittava ennen kanavan käyttämistä ja vapautettava jälkeenpäin, ja monimutkaisten, useiden muiden komponenttien kanssa toimivien metodien toteutuksessa on pidettävä kirjaa siitä, mitä lukkoja metodilla on milloinkin hallussaan lukkiutumisen ja saman lukon moninkertaisesta lukitsemisesta johtuvien virheiden välttämiseksi.

Myöskään Stage-moottoria käytettäessä rinnakkaisuutta ei voi jättää täysin huomiotta. Esimerkiksi fysiikkamoottoria toteutettaessa olio A saattoi käsitellä törmäystä olio B:n kanssa samanaikaisesti kuin olio B käsitteli törmäystä olio A:n kanssa, jolloin sama törmäys käsiteltiin kahdesti. Ratkaisuna fysiikkakomponentit pitävät yllä listaa niiden olioiden osoitteista, joiden kanssa ne ovat törmänneet nykyisen ruudunpäivityksen aikana, ja törmäyksen käsittely pysäytetään, mikäli toinen osapuoli on jo käsitellyt sen. Yleisesti ottaen Stage-moottoria käytettäessä tavallisista rinnakkaisuusongelmista ei kuitenkaan tarvitse huolehtia läheskään yhtä usein kuin lukkopohjaisessa vertailumoottorissa, koska aktorin metodeja suoritetaan aina vain yhdestä säikeestä kerrallaan. Tällöin komponenttien sisäistä tilaa ei tarvitse suojata lukoilla tai muilla rinnakkaisuudenhallintamekanismeilla. Tämän vuoksi Stage-moottorin päälle toteutettujen ohjelmien ohjelmointivirheet ilmenevät yleensä deterministisesti, minkä ansiosta ne on useimmi-

ten helpompi löytää ja korjata kuin Stage 11 -moottorissa avulla toteutetuissa ohjelmissa esiintyvät rinnakkaisuusongelmat. Itse ohjelmointi Stage-moottorin pohjalta on sen sijaan hiukan työläämpää aktorijärjestelmän aiheuttaman lisätyön vuoksi: kaikki viestinkäsittelijämetodit on rekisteröitävä ennen käyttöä, viestien sisältämät parametrit on pakattava viestiolioiden sisään ja jos viestin käsittely vaatii uusien viestien lähettämistä, viestiketjun tilaa hallinnoimaan tarvitaan tapahtumakonteksti. Lisäksi tällainen useiden viestien pituinen laskenta on pilkottava useiksi erillisiksi viestinkäsittelijämetodeiksi, mikä heikentää hieman koodin ymmärrettävyyttä ja hankaloittaa eri suorituspolkujen seuraamista.

Suurin ongelma pelien toteuttamisessa Stage-moottorilla on kuitenkin se, että kaikkien viestinkäsittelijöiden suorituspolkujen on päätyttävä vastausviestin lähettämiseen alkuperäisen viestin lähettäjälle – jos jokin käsittelijä jättää vastausviestin lähettämättä, pelimoottori lukkiutuu pelisilmukan jäädessä odottamaan käsittelijän vastausta. Tämä voidaan kuitenkin estää muuttamalla moottorin pohjana toimivan aktorijärjestelmän toteutusta siten, että pääohjelmasäie jää vastausviestin sijaan odottamaan, että aktorijärjestelmän kaikki työntekijäsäikeet ovat lepotilassa, jolloin kaikki nykyisen pelisilmukan suoritusvaiheen viestit on käsitelty. Tällöin myös moottorin toiminta tehostuisi jokaisen ruudunpäivityksen aikana lähetettävien viestien ja niistä kirjaa pitävien tapahtumakontekstien määrän laskiessa. Stage 11 -moottorin säiealtaan toiminta perustuu juuri tähän ratkaisuun, mutta Stage-moottorissa se olisi vaatinut muutoksia Theron-kirjaston toteutukseen.

Skaalautuvuus

Toteutusten skaalautuvuutta mitattiin suorittamalla demo-ohjelmaa, jossa 100 palloa törmäili laatikossa, jonka koko oli 20 pallon sädettä. Testilaitteistossa oli neliytiminen Intel Core i7 -suoritin, jonka jokaista ydintä käsiteltiin Hyper Threading -tekniikan avulla kahtena loogisena ytimenä. Näiden loogisten ydinten avulla suoritin kykenee suorittamaan samanaikaisesti kahdeksaa laitteistosäiettä, joskaan neljästä virtuaaliytimestä saatava suorituskykyhyöty ei ole yhtä suuri kuin varsinaisista fyysistä ytimistä [MaD02]. Toisin sanoen hyvin skaalautuvan rinnakkaisohjelman suorituskyvyn pitäisi testilaitteistolla nousta säikeiden määrää kasvatettaessa aina kahdeksaan säikeeseen asti, mutta yli neljästä säikeestä saatava suorituskykyetu on melko pieni.

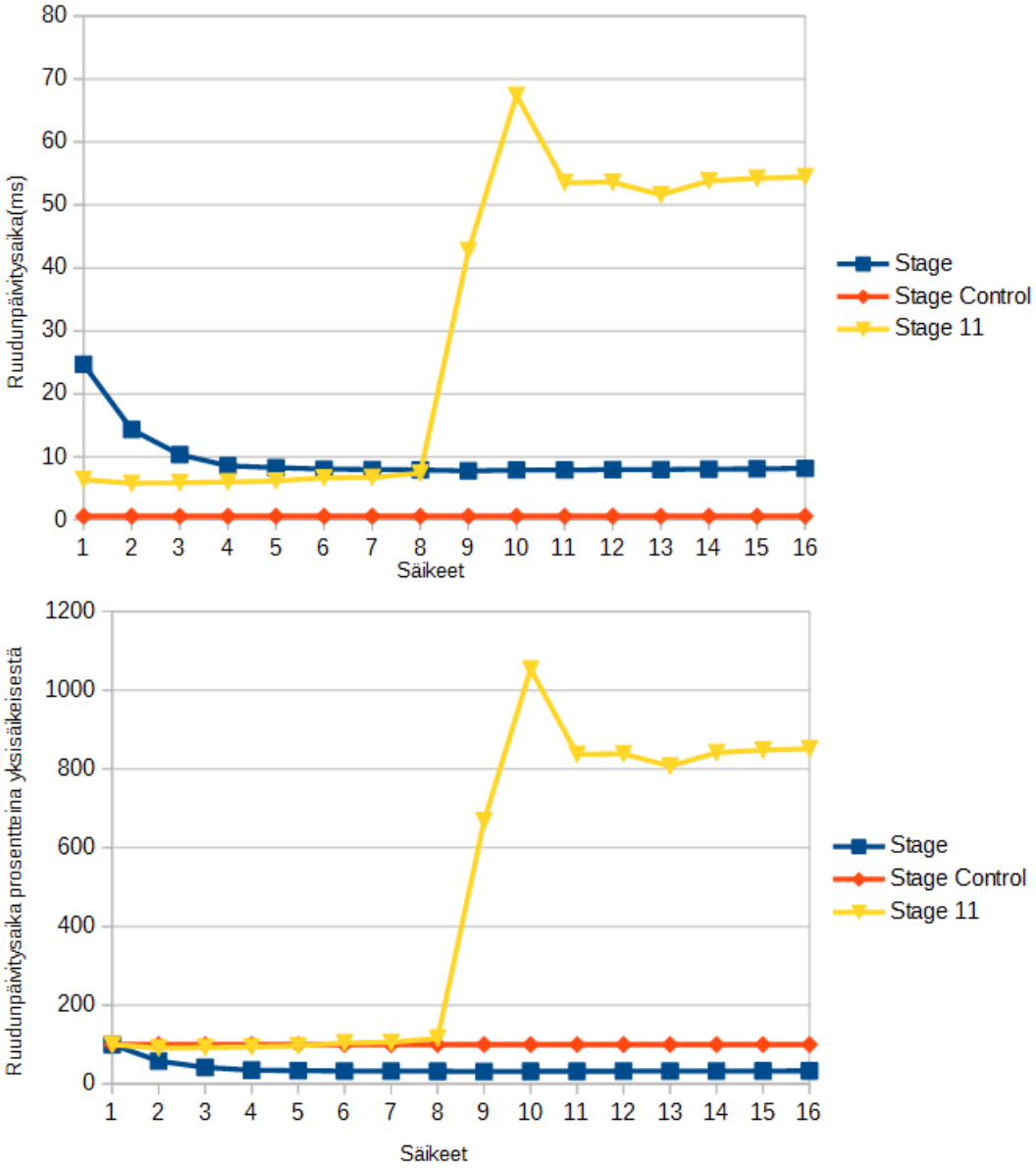
Demoa suoritettiin eri määrällä säikeitä ja suorituksen jälkeen kirjattiin ylös ruudunpäi-

	Stage			Stage Control			Stage 11		
Säikeitä	minimi	maksimi	keskiarvo	minimi	maksimi	keskiarvo	minimi	maksimi	keskiarvo
1	24,65	24,74	24,71	0,59	0,6	0,6	6,37	6,41	6,4
2	14,29	14,4	14,34	0,59	0,6	0,6	5,79	5,84	5,83
3	10,32	10,42	10,37	0,59	0,6	0,6	5,74	6,05	5,92
4	8,52	8,67	8,58	0,59	0,6	0,6	5,98	6,07	6,02
5	8,25	8,37	8,31	0,59	0,6	0,6	6	6,27	6,18
6	8,01	8,15	8,05	0,59	0,6	0,6	6,51	6,88	6,68
7	7,86	8,06	8	0,59	0,6	0,6	6,65	6,81	6,76
8	7,76	8	7,93	0,59	0,6	0,6	7,34	7,46	7,41
9	7,71	7,92	7,77	0,59	0,6	0,6	38,97	49,34	42,74
10	7,79	7,97	7,9	0,59	0,6	0,6	66,64	68,67	67,37
11	7,82	8	7,91	0,59	0,6	0,6	52,37	54,61	53,54
12	7,89	8,06	8	0,59	0,6	0,6	50,61	57,37	53,62
13	7,93	8,07	8	0,59	0,6	0,6	49,14	52,77	51,62
14	7,99	8,12	8,07	0,59	0,6	0,6	53,25	54,82	53,82
15	8,04	8,18	8,09	0,59	0,6	0,6	53,51	54,83	54,23
16	8,15	8,25	8,19	0,59	0,6	0,6	52,35	56,32	54,45

Taulukko 6.1: Keskimääräinen ruudunpäivitysaika millisekunteina eri säiemäärillä

vitykseen keskimäärin kulunut aika. Koska pallot lähtevät demon alussa liikkeelle satunnaisista sijainneista satunnaisiin suuntiin ja muut tietokoneen taustaprosessit saattoivat hidastaa suoritusta satunnaisesti, mittaustulokset vaihtelivat hiukan eri suorituskerroilla. Näiden satunnaisvaikutusten minimoimiseksi jokaista moottori-säiemääräyhdistelmää testattiin viisi kertaa, minuutti kerrallaan, ja jokaisen testin jälkeen kirjattiin ylös suorituskerran keskimääräinen ruudunpäivitysaika millisekunteina. Poikkeuksena Stage Control -toteutuksen viittä mittausta ei toistettu muilla säiemäärillä, sillä Stage Control on yksisäikeinen, eikä säiemäärä siksi vaikuta sen suorituskykyyn. Taulukkoon 6.1 on merkitty eri moottori-säiemääräyhdistelmien pienin ja suurin keskimääräinen ruudunpäivitysaika, sekä kaikkien suorituskertojen keskiarvot. Kaaviossa 6.1 näkyvät suorituskertojen tulosten keskiarvot.

Testeissä osoittautui, että vaikka Stage-moottori oli yhdellä säikeellä huomattavasti muita toteutuksia raskaampi, se myös skaalautui useille säikeille paljon lukkopohjaista Stage 11 -toteutusta paremmin. Stage-moottorin suorituskyky parani säiemäärän kasvaessa aina yhdeksään säikeeseen saakka, jolloin ruudunpäivitykseen kulunut aika oli noin 31,5% prosenttia siitä, mitä se oli yhdellä säikeellä. Tämän jälkeen suorituskyvyn heikentyminen oli hyvin hidasta, ja vielä 16 säikeellä ruudunpäivitysaika oli noin 33,2% yksisäikeisen suorituksen ajasta. Stage 11 puolestaan saavutti parhaimman tuloksensa kahdella säikeellä, jolloin suoritusaika laski alle 10 prosenttia yksisäikeiseen suoritukseen verrattuna. Säiemäärän kasvaessa kuuteen suoritus oli jo hitaampaa kuin yhdellä



Kaavio 6.1: Pelimoottoritoteutusten skaalautuvuus useille säikeille

säikeellä ja kun ohjelmistosäikeitä oli enemmän kuin laitteistosäikeitä, suorituskyky romahti täysin, todennäköisesti johtuen siitä, että kaikkia työntekijäsäikeitä ei voitu pitää suoritustilassa samanaikaisesti, jolloin prosessori joutui suorittamaan raskaita säikeenvaihto-operaatioita nopeasti ja usein.

Stage 11 -moottorin huono skaalautuminen sekä se, että yksisäikeinen Stage Control -moottori toimi erittäin paljon muita toteutuksia nopeammin osoittaa, että monisäikeiset toteutukset olivat yksinkertaisen demo-ohjelman kannalta aivan liian hienojakoisia. Demon toteutuksessa käytetyt yksinkertaiset pelioliokomponentit olivat liian kevyitä, jotta rinnakkaistamisesta olisi saatu mitään hyötyä – eri säikeiden ja komponenttien keskinäiseen viestintään ja synkronointiin kului paljon enemmän aikaa kuin varsinaisen pelitilan

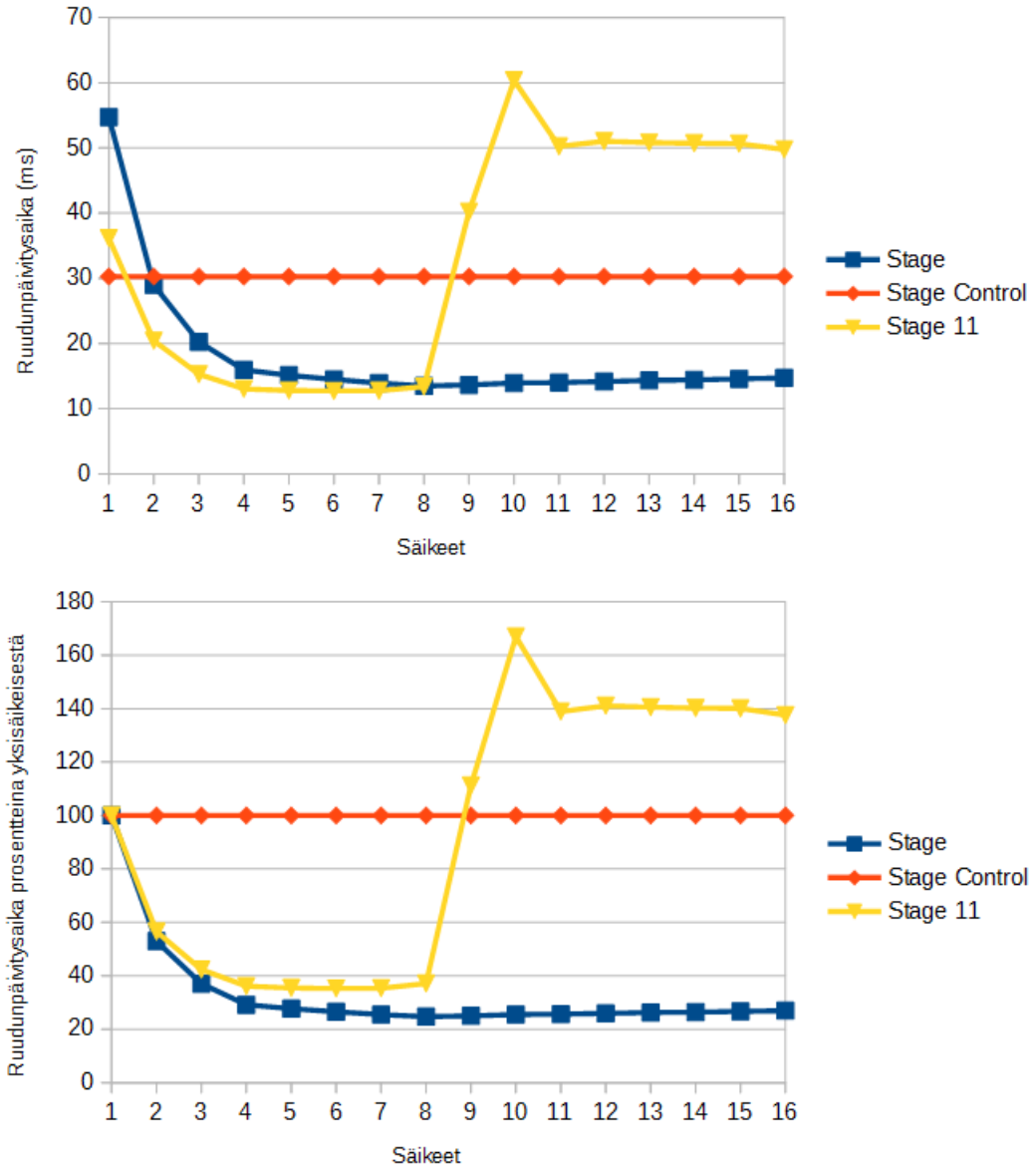
laskentaan. Jotta saataisiin tietoa siitä, miten hyvin toteutukset toimivat hyvin raskaan laskennan kanssa, jokaiseen demo-ohjelman palloon liitettiin Waiter-komponentti, joka laskee alkulukuja tiettyyn rajaan asti. Tällä simuloitiin rinnakkaislaskennan ideaalitapausta: laskentaa, joka vie paljon aikaa, muttei vaadi eri säikeiden välistä kommunikointia tai synkronointia. Mittaukset toistettiin siten, että jokainen Waiter-komponentti laski 1000 ensimmäistä alkulukua jokaisen ruudunpäivityksen aikana, ja tulokset näkyvät taulukossa 6.2 sekä kaaviossa 6.2.

Laskennan muuttaminen raskaammaksi paransi selvästi molempien rinnakkaistoteutusten skaalautuvuutta. Stage oli nyt tehokkaimmillaan kahdeksalla säikeellä, jolloin suoritukseen kului 24,7% siitä, mitä siihen kului yhdellä säikeellä, eikä säiemäärän kasvattaminen tästä aiheuttanut edelleenkään moottorin merkittävää hidastumista: 16:lla säikeellä nousi 26,9%:iin yksisäikeisestä. Koska laitteistossa oli neljä fyysistä ydintä ja suoritusaika oli parhaimmillaan alle neljäsosa yksisäikeisestä, Stage skaalautui lähes niin hyvin, kuin sen oli teoriassa mahdollista. Stage 11 saavutti nyt tehokkuushuippunsa kuudella säikeellä, jolloin sen ruudunpäivitysaika oli noin 35,2% yksisäikeisestä, mutta säiemäärän nostaminen yhdeksään sai edelleen moottorin suorituskyvyn romahtamisen.

Aktoripohjainen toteutus skaalautui siis useille säikeille kaikissa mittauksissa paremmin kuin lukkopohjainen: vaikka se oli toteutuksista selvästi raskain, se myös paransi tulostaan eniten säikeiden määrän kasvaessa, eikä sen suorituskyky juurikaan heikentynyt säiemäärän noustessa optimista.

	Stage			Stage Control			Stage 11		
Säikeitä	minimi	maksimi	keskiarvo	minimi	maksimi	keskiarvo	minimi	maksimi	keskiarvo
1	54,64	54,96	54,74	30,27	30,29	30,28	36,14	36,21	36,17
2	28,93	29,03	28,98	30,27	30,29	30,28	20,27	20,64	20,44
3	20,23	20,3	20,27	30,27	30,29	30,28	15,21	15,44	15,31
4	15,89	16,03	15,96	30,27	30,29	30,28	13	13,15	13,06
5	15,08	15,2	15,15	30,27	30,29	30,28	12,7	12,87	12,81
6	14,41	14,59	14,51	30,27	30,29	30,28	12,56	13,03	12,74
7	13,82	14,05	13,95	30,27	30,29	30,28	12,59	13,02	12,77
8	13,38	13,65	13,5	30,27	30,29	30,28	13,21	13,73	13,41
9	13,63	13,73	13,67	30,27	30,29	30,28	33,62	52,14	40,22
10	13,87	14,03	13,96	30,27	30,29	30,28	59,38	62,4	60,38
11	13,95	14,15	14,03	30,27	30,29	30,28	48,72	52,27	50,24
12	14,09	14,28	14,19	30,27	30,29	30,28	50,34	52,46	51,02
13	14,29	14,4	14,35	30,27	30,29	30,28	49,28	52,25	50,82
14	14,36	14,48	14,44	30,27	30,29	30,28	48,34	52,06	50,71
15	14,5	14,68	14,59	30,27	30,29	30,28	49,91	50,99	50,66
16	14,7	14,77	14,74	30,27	30,29	30,28	48,07	51,94	49,77

Taulukko 6.2: Keskimääräinen ruudunpäivitysaika millisekunteina Waiter-komponentin kanssa



Kaavio 6.2: Pelimoottoritoteutusten skaalautuvuus Waiter-komponentin kanssa

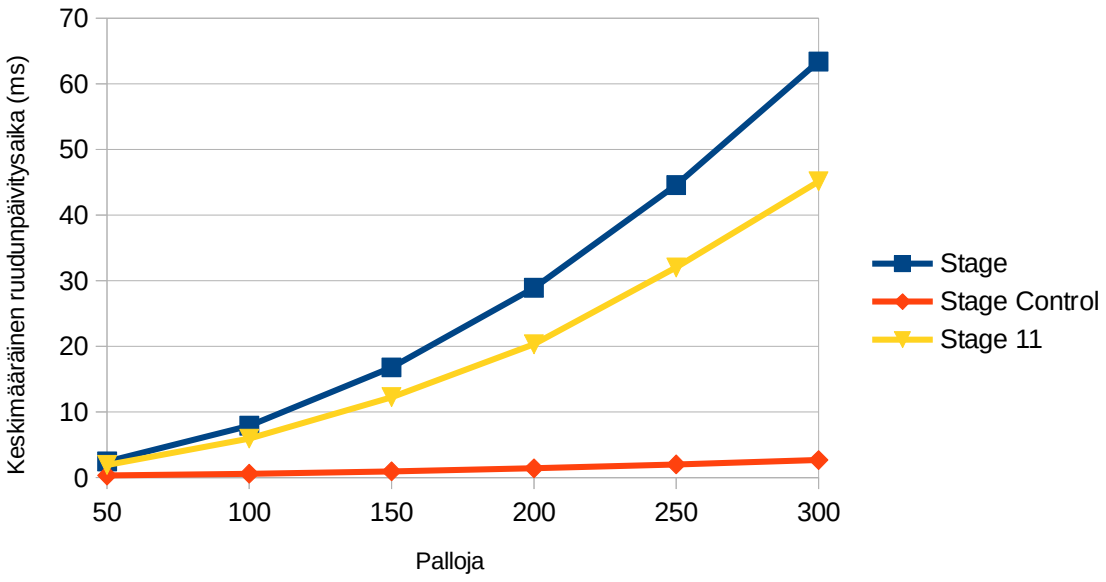
Tehokkuus

Toteutuksen tehokkuutta mitattiin suorittamalla demo-ohjelmaa vaihtelevalla määrällä palloja: ensimmäisessä testissä palloja oli 50 ja laatikon koko oli 15, ja jokaisen testin jälkeen pallojen määrää kasvatettiin 50:llä ja laatikon kokoa viidellä. Moottorit asetettiin käyttämään sitä säiemäärää, jolla ne olivat saaneet parhaat tulokset skaalautuvuustesteissä ja tehokkuutta mitattiin joka pallomäärällä viisi kertaa, minuutti kerrallaan. Taulukossa 6.3 näkyvät eri suorituskerroilla saatujen keskimääräisten ruudunpäivitysaikojen pienin ja suurin tulos, sekä kaikkien tulosten keskiarvot. Kaavioon 6.3 on merkitty suorituskertojen keskiarvot.

Stage oli moottoreista selvästi raskain, mutta sen ruudunpäivitykseen kuluttama aika oli

Laatikon koko	Pallot	Moottori	minimi	maksimi	keskiarvo
15	50	Stage	2,44	2,49	2,47
20	100		7,88	7,92	7,91
25	150		16,69	16,93	16,79
30	200		28,27	29,25	28,91
35	250		43,8	45,23	44,57
40	300		62,16	64,21	63,39
15	50	Stage Control	0,33	0,33	0,33
20	100		0,59	0,59	0,59
25	150		0,95	0,96	0,96
30	200		1,42	1,43	1,43
35	250		2	2,01	2,01
40	300		2,67	2,71	2,69
15	50	Stage 11	1,92	1,95	1,94
20	100		5,86	6,13	5,95
25	150		11,85	12,47	12,25
30	200		19,82	20,81	20,32
35	250		30,45	33,44	32,02
40	300		44,03	46,55	45,13

Taulukko 6.3: Ruudunpäivitysaika millisekunteina suhteessa peliolioiden määrään



Kaavio 6.3: Pelimoottoritoteutusten tehokkuus

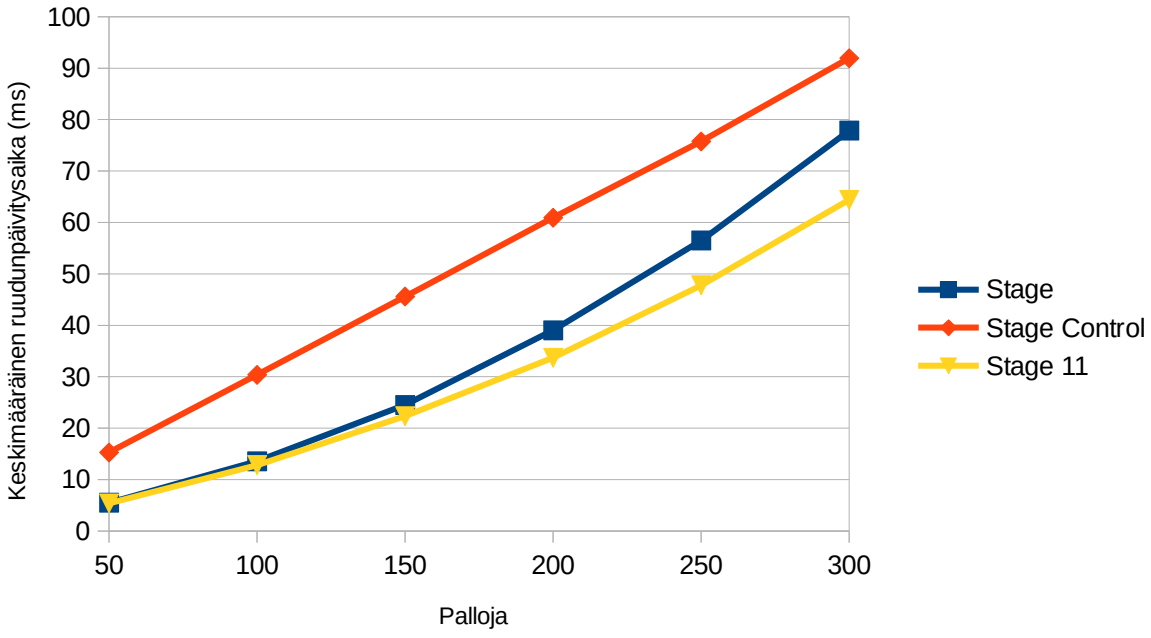
kuitenkin samaa kokoluokkaa kuin Stage 11:n. Yksisäikeinen Stage Control oli kolmesta moottorista nopeudeltaan täysin omassa luokassaan, johtuen skaalautuvuustulosten yhteydessä mainitusta liiasta hienojakoisuudesta. Moottoreiden yksinkertaiset komponenttitoteutukset olivat niin kevyitä, että molemmat rinnakkaistoteutukset käyttivät valtaosan ajastaan rinnakkaisuuden hallintaan ja tilan synkronointiin. Tämän vuoksi tehokkuustestit toistettiin skaalautuvuustulosten yhteydessä kuvatun Waiter-komponentin kanssa, siten että jokainen pallo laski ensimmäiset 1000 alkulukua. Tulokset ovat taulu-

kossa 6.4 ja kaaviossa 6.4.

Raskaan laskennan kanssa Stage Control oli odotetusti moottoreista hitain ja Stage 11 nopein. Kun peliolioita oli vähän, Stage oli lähestulkoon yhtä nopea kuin Stage 11, mutta peliolioiden määrän kasvaessa Stagen suorituskyky heikkeni lukkopohjaista toteutusta nopeammin ja oli 300 pallolla noin 37% Stage 11:a hitaampi. Vielä tällöinkin

Laatikon koko	Pallot	Moottori	minimi	maksimi	keskiarvo
15	50	Stage	5,48	5,53	5,5
20	100		13,42	13,69	13,56
25	150		23,62	24,82	24,51
30	200		38,92	39,15	39,04
35	250		55,68	57,31	56,5
40	300		76,53	78,95	77,86
15	50	Stage Control	15,26	15,29	15,27
20	100		30,36	30,4	30,38
25	150		45,55	45,64	45,6
30	200		60,9	60,97	60,92
35	250		73,31	76,38	75,76
40	300		91,85	92,06	91,94
15	50	Stage 11	5,36	5,42	5,39
20	100		12,77	12,88	12,82
25	150		22,15	22,42	22,33
30	200		33,49	34,07	33,69
35	250		47,4	48,54	47,77
40	300		63,68	64,92	64,4

Taulukko 6.4: Ruudunpäivitysaika millisekunteina suhteessa peliolioiden määrään Waiter-komponentin kanssa



Kaavio 6.4: Pelimoottoritoteutusten tehokkuus Waiter-komponentin kanssa

Stage oli yksisäikeistä Stage Control -moottoria tehokkaampi, mutta hyvin suurilla peliolio- ja komponenttimäärillä viestien hallinnoimiseen kuluva ylimääräinen aika saattaa muodostua ongelmaksi. Tätä voidaan mahdollisesti lievittää vähentämällä moottorin lähettämien viestien määrää, esimerkiksi muuttamalla aktorijärjestelmän toteutusta siten, että pelisilmukan laskentavaiheen päättyminen saadaan selville ilman vastausviestejä (ks. luku 6.2, Erikoistamisvaikeus), tai käyttämällä karkeajakoisempaa rinnakkaistamismallia (ks. luku 5.2). Esimerkiksi mikäli voitaisiin tehokkaasti taata, ettei saman peliolion eri komponentteja suoriteta samanaikaisesti eri säikeistä, näiden komponenttien sisältä voitaisiin kutsua muiden komponenttien palveluita suoraan, sen sijaan että jouduttaisiin Stage-moottorin tapaan käyttämään raskaampaa viestinvälitystä. Tämä saattaisi heikentää moottorin skaalautuvuutta silloin, kun peliolioita on vähän, mutta peliolioiden määrän ollessa hyvin pieni moottori toimii yleensä tarpeeksi nopeasti, ettei suorituskyvyn lasku käytännössä näy käyttäjälle.

6.3 Aktoreiden soveltuvuus pelimoottoreihin

Tässä luvussa pohditaan, miten aktorimallin pelimoottoreihin lupaamat hyödyt (ks. luku 4.4) toteutuvat käytännössä ja onko aktorijärjestelmä todellisuudessa järkevä pohja pelimoottorille.

Suorituskyky

Stage osoittautui testeissä perinteisillä lukkopohjaisilla rinnakkaistamisratkaisuilla toteutettua vertailumoottoria paremmin skaalautuvaksi. Tämän perusteella vaikuttaa, että aktoripohjaisella lähestymistavalla kyetään saavuttamaan perinteisiä lukkopohjaisia ratkaisuja parempi suorituskyky, kunhan käytössä on vain riittävän monta suoritinydintä. Testilaitteiston neljä ydintä ei kuitenkaan vielä riittänyt nostamaan Stage-moottorin suorituskykyä lukkopohjaisen Stage 11-vertailutoteutuksen tasolle saati sen ohi, joten mahdollisten suorituskykyetujen saavuttamiseksi aktoripohjainen moottori vaatii testitulosten perusteella vähintäänkin kahdeksanytimistä suoritinta. Tämän vuoksi aktoripohjaiset ratkaisut vaikuttavat hyödyllisiltä, mikäli pelien alustoina toimivien laitteiden suoritinydinten määrä jatkaa tulevaisuudessa kasvuaan, mutta aktorimallia on vielä hankala suositella markkinoilla nykyisin oleville alustoille tähtääville peleille ja pelimoottoreille. Koska testilaitteistossa oli vain neljä suoritinydintä, Stage-moottorin skaalautuvuutta ja suorituskykyä olisi hyvä tulevaisuudessa testata suurempaa laitteisto-

säiemäärää tukevalla alustalla. Näin saataisiin parempi kuva aktorimallin vaikutuksesta pelimoottorin tehokkuuteen erityisesti erittäin monia prosessoriytimiä sisältävillä laitteistoalustoilla sekä siitä, saavuttaako aktoripohjainen pelimoottoritoteutus todellisuudessa lukkopohjaisen suorituskykyä, kun suoritinydinten määrää kasvatetaan, ja kuinka monta ydintä tähän käytännössä vaaditaan.

Hienojakoisuus

Eräs aktorimallin hyödyistä oli aktorijärjestelmien ja pelimoottorien käyttämien abstraktioiden samankaltaisuus: videopelien pelimaailmat koostuvat itsenäisistä peliolioista, jotka toimivat omien käyttäytymismalliensa mukaan ja ovat ajoittain vuorovaikutuksessa keskenään, aivan kuin aktorimallin aktorit. Tämän ansiosta moottorin rinnakkaistaminen on melko yksinkertaista, eikä pelin toteuttajan tarvitse juurikaan huolehtia rinnakkaisuusongelmista, sillä yksittäisten pelioliokomponenttien sisällä suoritus on aina yksisäikeistä. Samalla ongelmaksi nousee kuitenkin se, että yksittäisiä peliolioita ja niiden komponentteja rinnakkaisuuden yksikkönä käyttävä pelimoottori on erittäin hienojakoinen, eli yksittäiset, rinnakkain suoritettavat ohjelman osat ovat hyvin pieniä. Tämä saattaa aiheuttaa suorituskykyongelmia, koska tällöin varsinaiseen laskentaan käytetään vain vähän aikaa rinnakkaisuudenhallintaan verrattuna.

Nämä suorituskykyongelmat nousivat esille tehokkuustesteissä, joissa rinnakkaisuudenhallinnan vaatima työ sai sekä Stage-moottorin että yhtä hienojakoisen Stage 11 -vertailumoottorin toimimaan huomattavasti yksisäikeistä Stage Control -moottoria hitaammin. Tulosten pääsyynä oli todennäköisesti testeissä käytettyjen komponenttien yksinkertaisuus: kun varsinaiseen pelitilan päivittämiseen tarvittava laskenta on kevyttä, rinnakkaisuuden hallinta vie siihen verrattuna paljon aikaa. Siksi olisikin tulevaisuudessa järkevää toteuttaa Stage-moottorin avulla pelimoottorikomponentteja, jotka vastaavat monimutkaisuudeltaan ja laskennalliselta raskaudeltaan kaupallisissa pelimoottoreissa olevia – näin saataisiin tarkempaa tietoa siitä, miten suuri haitta aktoripohjaisen lähestymistavan hienojakoisuudesta todellisuudessa aiheutuu ja soveltuuko aktoripohjainen pelimoottori käytännössä nykypelien pohjaksi.

Hienojakoisuutta voisi vähentää myös käyttämällä pienimpänä rinnakkaisuuden yksikkönä peliolioita pelioliokomponenttien sijaan, mutta Stage-moottorista tämän testaamista varten kehitetty karkeajakoisempi versio osoittautui käytännössä normaalia Stage-moottoria hitaammaksi johtuen lisätyöstä, joka vaadittiin viestien ohjaamiseksi

oikeille vastaanottajakomponenteille peliolion sisällä. Jotta tämä lähestymistapa parantaisi suorituskykyä käytännössä, se vaatisi todennäköisesti muutoksia aktorijärjestelmän toimintaan – jos esimerkiksi voidaan varmistaa tehokkaasti, että saman peliolion eri komponenttiaktoreita ei suoriteta rinnakkain, kyettäisiin niiden sisältä kutsumaan muiden saman peliolion komponenttien metodeja suoraan, jolloin lähetettävien viestien määrä pienenisi ja viestiä kohden suoritettava varsinaisen työn määrä kasvaisi, pienentäen järjestelmän hienojakoisuutta.

Toinen vaihtoehto olisi vieläkin karkeajakoisempi ratkaisu, jossa pelimoottorin pienin rinnakkaisuusyksikkö olisi kokonainen pelimoottorimoduuli. Tällöin aktoreita olisi vain pelimoottorimoduulien verran ja moduulit olisivat vuorovaikutuksessa toistensa kanssa viestinvälityksen avulla. Tällöin kuitenkin menetettäisiin aktoripohjaisuuden suurimmat edut – hyvä skaalautuvuus useille suoritinytimille ja abstraktioiden yhteensopivuus – joten juuri aktorimallin käyttöä muiden rinnakkaisuudenhallintamekanismien sijaan olisi vaikeampi perustella.

Tapahtumajärjestelmä

Aktoripohjaisuus todellakin helpottaa pelimoottorin viesti- tai tapahtumajärjestelmän toteuttamista, sillä aktorikirjasto tarjoaa valmiin toiminnallisuuden viestien luomista, lähettämistä ja käsittelyä varten. Pelimoottorien normaaleista tapahtumajärjestelmistä poiketen viestin lähettäjän on tunnettava vastaanottajan osoite, mutta tämä helppo korjata luomalla tapahtumakanava-aktori, joka välittää vastaanottamansa viestit niistä kiinnostuneille vastaanottajille. Tällöin tapahtumajärjestelmä toimii käytännössä aivan kuten muissakin pelimoottoreissa: viesteistä kiinnostunut olio rekisteröityy ottamaan vastaan tietyn tyyppisiä viestejä, minkä jälkeen sille ohjataan automaattisesti kaikki tapahtumajärjestelmään syötetyt kyseistä tyyppiä olevat viestit.

Stage-moottorin tapahtumajärjestelmän suurin ongelma on se, että Theron-kirjaston viestinvälitysmekanismit perustuvat C++-kielen malleihin (templates), minkä vuoksi kaikki pelin viestit ja viestinkäsittelijät on tunnettava jo käännösaikana. Tämä olisi todellisessa pelimoottorissa ongelma, sillä monet pelien toiminnoista ja mekaniikoista halutaan yleensä toteuttaa suoritusaikana tulkattavina skripteinä, jolloin pienet peliin tehtävät muutokset eivät vaadi koko ohjelman uudelleenkääntämistä. Stage-moottoriin olisi täysin mahdollista lisätä skriptikomponenttiluokka eli peliolioon liitettävä komponentti, joka toimii rajapintana aktorijärjestelmän ja jollakin muulla kielellä toteutetun

skriptin välillä, mutta tämä komponentti pystyisi kuitenkin viestimään muiden aktorien kanssa pelkästään peliin käännösvaiheessa määriteltyjen viestien avulla, mikä tekisi eri skriptien välisestä vuorovaikutuksesta hankalaa.

Mahdollinen ratkaisu ongelmaan voisi olla erityisen ”skriptiviestitietueen” käyttäminen. Skriptiviesti on tavallinen Event-viestitietue, joka sisältää mielivaltaista binääridataa sisältävän parametrikentän sekä metadatakentän, joka ilmaisee parametridatan tietotyypin. Tämä mahdollistaa eri skriptien mielivaltaisen vuorovaikutuksen: kun skripti lähettää itse määrittelemänsä mielivaltaisen viestin, skriptiä pelimoottorissa suorittava skriptikomponentti paketoi viestin skriptiviestin sisään ja lähettää sen aktorijärjestelmän kautta vastaanottajaskriptiä suorittavalle skriptikomponentille, joka purkaa alkuperäisen viestin skriptiviestistä ja välittää sen skriptilleen. Tämä on tietenkin sisäänrakennettujen viestien käyttämistä tehottomampaa, mutta skriptit eivät yleensä ole muutoinkaan yhtä tehokkaita kuin varsinainen käännetty peliohjelma, ja suorituskyvylle kriittiset komponentit voidaan tietenkin tarvittaessa toteuttaa itse peliohjelman osana.

On myös tärkeä huomata, että skriptikomponenttien toteuttamisen hankaluus Stage-moottorissa johtuu Stage-moottorin pohjana toimivan Theron-kirjaston toteutuksesta, eikä ongelmaa siksi välttämättä esiinny lainkaan muihin aktorikirjastoihin perustuvissa moottoreissa – itse asiassa Adam Lake kuvailee kirjassaan *Game Programming Gems 8* moottorin, jossa Lua-kielellä toteutetut skriptit kykenevät viestimään suoraan toistensa kanssa C++-kielellä toteutetun aktorijärjestelmän kautta [LaA10, sivut 480-482]. Myös tämän vuoksi ennen varsinaisen pelimoottorin toteuttamista aktorijärjestelmän avulla olisi hyvä tehdä tarkempaa jatkotutkimusta saatavilla olevista aktorikirjastoista ja verrata niiden ominaisuuksia nimenomaan pelimoottorin toteuttamisen näkökulmasta.

Rinnakkaisuudenhallinnan vaikeus

Eräs aktoripohjaisuuden lupaamista eduista oli se, että aktoriohjelman toteuttajan ei tarvitse huolehtia yhtä paljon perinteisten rinnakkaistamismekanismien mukanaan tuomista ongelmista kuten lukkiutumisista ja kilpatilanteista. Tämä toteutuu melko hyvin myös käytännössä, mutta aktoripohjaisessa moottorissakaan rinnakkaisuutta ei voi jättää täysin huomiotta. Lisäksi mikäli aktorien pohjalta ryhdytään kehittämään varsinaista kaupallisten pelimoottorien kanssa kilpailukykyistä pelimoottoritoteutusta, ohjelmakoodissa joudutaan todennäköisesti suorituskyvyn optimoimiseksi käyttämään paikoittain perinteisiä rinnakkaisuudenhallintamekanismeja, mikä lisää rinnakkaisuus-

ongelmien esiintymisen riskiä.

Tästä huolimatta aktoripohjainen lähestymistapa vähentää Stage-moottorin demo-ohjelmaa toteutettaessa saatujen kokemusten perusteella rinnakkaisuusongelmien esiintymistä huomattavasti ja tekee niiden löytämisestä ja korjaamisesta helpompaa, koska yksittäisten aktorien sisällä ohjelmakoodi on lähtökohtaisesti yksisäikeistä. Kääntöpuolena aktorikoodin kirjoittaminen on työläämpää ja monimutkaisten viestiketjujen tapauksessa ohjelman kulkua on hankalampi seurata. Koska tämä johtuu pääasiassa siitä, ettei C++-kieltä ole alun perin suunniteltu aktorimallin ja asynkroniseen viestinvälitykseen perustuvien ohjelmien tarpeisiin, ongelma voitaisiin periaatteessa ratkaista kehittämällä uusi ohjelmointikieli, joka abstrahoi viestinvälityksen työläämmät osat näkymättömiin ja joko kääntyy C++-koodiksi tai toimii pelimoottorin päällä skriptikielenä. Jos tämä ei käy päinsä, ongelmaa voidaan mahdollisesti lievittää myös esimerkiksi hyvillä ohjelmointi- ja kommentointikäytännöillä, jotka dokumentoivat selvästi viestiketjujen rakenteen, korvaamalla viestinkäsittelyssä usein toistuvia rakenteita makroilla tai toteuttamalla viestiketjut vuorottaisrutiineina.

Suurin haaste pelimoottorin ja aktorimallin yhdistämisessä on se, että aktorimalli on suunniteltu täysin asynkroniseksi: aktorijärjestelmälle annetaan tehtäviä, jotka etenevät omaan tahtiinsa muiden tehtävien kulusta juurikaan välittämättä. Pelimoottori on kuitenkin luonnostaan synkroninen: sen pelisilmukka jakautuu vaiheisiin, eikä seuraavaan vaiheeseen saatikka silmukan seuraavaan suorituskertaan voida edetä, ennen kuin nykyisen vaiheen kaikki laskenta on saatu päätökseen. Aktoripohjaisessa järjestelmässä ongelmaksi nousee se, miten moottori tunnistaa kaiken laskennan päättyneen, kun laskentaa suorittavia aktoreita saattaa olla satoja tai jopa tuhansia. Mahdollisia ratkaisuja on useita, mutta Stage-moottorissa on käytetty sitä, joka asettaa vähiten rajoitteita aktorijärjestelmän ja itse pelin toteutukselle: aina kun aktori saa pelisilmukan yhden laskentavaiheen päätökseen, se lähettää laskennan päättymisestä kertovan AllDone-vastausviestin. Tämä yhdistettynä pelialueiden, niiden sisältämien peliolioiden ja peliolioiden omistamien komponenttien muodostamaan hierarkkiseen rakenteeseen mahdollistaa kaikkien pelimaailman aktorien laskennan seuraamisen ilman, että mikään yksittäinen aktori muodostuu pullonkaulaksi. Ratkaisun heikkous on se, että vastausviestien muodostaminen ja lähettäminen vie laskentatehoa ja että yksikin puuttuva vastausviesti voi johtaa jopa koko pelimoottorin pysähtymiseen. Todellisessa aktori-

moottorissa olisi tämän vuoksi järkevämpää käyttää samankaltaista ratkaisua, kuin Stage 11 -vertailumoottorin säiealtaassa: pelisilmukka seuraa suoraan työntekijäsäikeiden tilaa ja seuraavan vaiheen suoritus voidaan aloittaa, kun kaikki säikeet ovat lepotilassa eikä uusia viestejä ole enää odottamassa käsittelyä. Tämä kuitenkin todennäköisesti vaatii aktorikirjaston sisäisen toteutuksen muokkaamista tai jopa aktorikirjaston luomista varta vasten moottorin tarpeisiin, sillä useimmat aktorikirjastot eivät tarjoa suoraan palveluita säikeidensä tilan seuraamiseen.

Johtopäätökset ja jatkotutkimus

Aktorimallin avulla on mahdollista rakentaa pelimoottori, mutta jotta tämä olisi suorituskyvyn kannalta järkevää, on pelissä suoritettavan laskennan oltava riittävän raskasta, jotta rinnakkaisuuden hienojakoisuudesta johtuva lisätyö ei aiheuta laskennan merkittävää hidastumista, ja käytettävän laitteiston on tuettava riittävän montaa rinnakkaista laitteistosäiettä, jotta aktoripohjaisen laskennan hyvä skaalautuvuus tuottaa etuja muihin tekniikoihin verrattuna. Tätä tutkielmaa varten suoritettujen testien tulosten perusteella aktoripohjaisuus vaikuttaa skaalautuvuutensa vuoksi lupaavalta, mikäli suoritinydinten määrä kasvaa tulevaisuudessa entisestään, mutta monet nykyisistä laitteistoalustoista eivät vielä sisällä tarpeeksi rinnakkaisia laskentaytimiä muita rinnakkaistamistekniikoita vastaavan suorituskyvyn saavuttamiseksi.

Jotta saataisiin paremmin selville, miten hyvin aktoripohjaisuus soveltuu käytännössä nykyisten ja tulevien pelien tarpeisiin, jatkossa olisi hyödyllistä tutkia tarkemmin, missä olosuhteissa aktoripohjainen lähestymistapa toimii parhaiten. Käytännössä tämä tarkoittaisi Stage-moottorin suorituskyvyn tutkimista hyvin monia laitteistosäikeitä tukevassa laitteistossa, jotta saadaan selville, skaalautuuko aktoripohjainen moottori yhtä hyvin myös suurille säiemäärille vai alkaako suorituskyky heikkenemään tietyn pisteen jälkeen, sekä monimutkaisemman, kaupallisia pelimoottoreita vastaavan toiminnallisuuden toteuttaminen aktoripohjaisella moottorilla, jotta nähtäisiin, kuinka raskasta aktorijärjestelmän hallinnointi on nykypelien muuhun laskentaan verrattuna ja onko aktoripohjaisen moottorin hienojakoisuus ongelma myös käytännössä.

Toinen hyvä kohde jatkotutkimukselle olisi C++-kielelle saatavissa olevien erilaisten aktorikirjastojen ominaisuuksien vertailu erityisesti pelimoottorien näkökulmasta. Aktorijärjestelmän toteutustavasta riippuen tietyt pelimoottorin osat ja toiminnot kuten pelioliokomponenttien määrittely skriptikielillä saattavat olla helpompia toteuttaa. Li-

säksi aktorikirjastojen suorituskyvyssä saattaa olla eroja, millä voi olla suuri vaikutus erityisesti videopelien kaltaisissa tehokkuutta vaativissa sovelluksissa. Suorituskyvyn kannalta saattaisi myös olla kannattavaa tutkia aktorijärjestelmän räätälöimistä juuri pelimoottorin tarpeisiin – esimerkiksi jos aktorijärjestelmä kykenee suoraan ilmoittamaan pääsäikeelle laskennan päättymisestä, kun kaikki pelisilmukan yhden vaiheen viestit on käsitelty, aktoripohjainen moottori ei tarvitsisi vastausviestejä arvoa palauttamattomille viestinkäsittelijöille, mikä parantaisi moottorin suorituskykyä ja vakautta. Lisäksi monet kaupalliset pelimoottorit käyttävät hyvin matalan tason optimointeja kuten omia, nimenomaan pelien vaatimuksia varten suunniteltuja muistinhallintajärjestelmiään, joiden käyttöönotto aktoripohjaisessa pelimoottorissa vaatisi todennäköisesti aktorikirjaston toteutuksen muokkaamista.

Stage-pelimoottorin mielekkyyttä testattiin vertaamalla sen tehokkuutta ja skaalautuvuutta kahteen vertailumoottoriin, joista ensimmäinen oli yksisäikeinen ja toinen C++11:n säikeiden avulla rinnakkaistettu. Testit suoritettiin toteuttamalla kaikilla kolmella moottorilla sama yksinkertainen demo-ohjelma ja mittaamalla niiden keskimäärin ruudunpäivitykseen kuluttamaa aikaa samalla laitteistolla. Toteutuksen perusteella Stage-moottorin päälle toteutetuista ohjelmista oli helpompi tehdä virheettömiä kuin perinteisillä lukkopohjaisilla keinoilla rinnakkaistetuista ohjelmista, mutta niiden toteuttaminen oli myöskin työläämpää. Tehokkuudeltaan Stage pääsi melko lähelle lukkopohjaista vertailutoteutusta, joskin sen suorituskyky heikkeni nopeammin peliolioiden määrän kasvaessa. Skaalautuvuudeltaan se oli sen sijaan selvästi lukkopohjaista toteutusta parempi. Testitulosten perusteella aktorimalli vaikuttaa varteenotettavalta pohjalta pelimoottoreille tulevaisuudessa alustojen rinnakkaisuusasteen ja pelien käyttämän laskennan raskauden kasvaessa, mutta pienillä säiemäärillä se oli selvästi muita ratkaisuja hitaampi, mikä todennäköisesti aiheuttaisi ongelmia monilla markkinoilla nykyisin olevilla alustoilla. Jatkossa olisi järkevää tutkia tarkemmin aktoripohjaisten moottorien suorituskykyä enemmän reaalimaailman sovelluksia muistuttavissa tilanteissa, sekä aktorijärjestelmien räätälöimistä vastaamaan paremmin juuri pelimoottorien asettamia vaatimuksia.

7 Yhteenveto

Videopelit ovat aina pyrkineet ottamaan laitteiston tehoista kaiken irti, sillä ylimääräisten tehojen mahdollistamat parannukset esimerkiksi grafiikkaan tai pelin mittakaavaan voivat olla pelialalla merkittävä kilpailuetu. Kun tietokoneiden tehoja kasvatetaan pääasiassa lisäämällä rinnakkaisuutta, on tämän vuoksi tärkeää etsiä tehokkaita tapoja hyödyntää rinnakkaislaskentaa pelimoottoreissa. Ohjelmistojen rinnakkaistaminen ei kuitenkaan ole yksinkertaista, rinnakkaisuudenhallintamekanismit voivat huonosti toteutettuna aiheuttaa vakavia suorituskykyongelmia, virheitä tai jopa pelin lukkiutumisen. Eräs mahdollinen rinnakkaisuudenhallintaratkaisu on aktorimalli, joka perustuu itsenäisten toimijoiden väliseen asynkroniseen viestinvälitykseen.

Aktorimallin soveltuvuutta pelimoottoreihin tutkittiin toteuttamalla Stage-moottori. Stage on yksinkertainen aktoripohjainen 3D-pelimoottori, jossa jokainen peliolio ja peliolion komponentti on toteutettu aktorina ja niiden väliset vuorovaikutukset on mallinnettu aktorijärjestelmän viesteinä. Moottorin tehokkuutta ja skaalautuvuutta testattiin vertaamalla sitä sekä yksisäikeiseen että C++11:n säikeiden ja lukkojen avulla rinnakkaistettuun vertailumoottoritoteutukseen. Testitulosten perusteella aktoripohjainen pelimoottoritoteutus ei aivan yllä samaan suorituskykyyn kuin lukkopohjainen, mutta pääsee kuitenkin melko lähelle. Skaalautuvuudeltaan aktoripohjainen toteutus sen sijaan on selvästi lukkopohjaista parempi, minkä vuoksi tulevaisuuden erittäin monia prosessoriytimiä sisältävillä alustoilla aktorimalli voi ohittaa lukkopohjaisen ratkaisujen suorituskyvyn ja vaikuttaa siksi olevan tehokas tapa rinnakkaistaa pelimoottori. Jotta aktoripohjaisesta rinnakkaistamisesta olisi hyötyä, moottorin suorittaman laskennan on kuitenkin oltava riittävän raskasta – demo-ohjelman yksinkertaisuuden ja moottorin rinnakkaisuuden hienojakoisuuden vuoksi yksisäikeinen vertailutoteutus oli testeissä huomattavasti molempia rinnakkaistettuja toteutuksia tehokkaampi.

Skaalautuvuuden lisäksi toinen aktoripohjaisuuden merkittävä etu perinteisiin lukkopohjaisiin ratkaisuihin verrattuna on rinnakkaisuusongelmien välttäminen. Kun noudatetaan aktoriohjelmoinnin käytäntöjä, aktorimallia käytettäessä ei juurikaan tarvitse huolehtia tavallisista rinnakkaisuusongelmista kuten kilpatilanteista. Aivan täysin rinnakkaisuutta ei aktoriohjelmoinnissakaan voi jättää huomiotta ja pelin toteuttaminen aktoripohjaisen pelimoottorin päälle on jonkin verran työläämpää, mutta koska akto-

riohjelmoinnissa esiintyvät ohjelmointivirheet ovat pääosin determistisiä, niiden löytäminen ja korjaaminen on merkittävästi rinnakkaisuusongelmia helpompaa.

Aktoripohjaisuus siis vaikuttaa järkevältä lähestymistavalta pelimoottoriin silloin, kun laskenta on hyvin raskasta ja käytettävissä on hyvin suuri määrä suoritinytimiä. Ennen kuin aktorijärjestelmän päälle voidaan kuitenkaan toteuttaa varsinaista, nykyisten kaupallisten pelien pohjaksi kelpaavaa pelimoottoria, tarvitaan kuitenkin vielä lisätyötä. Stage-moottoria toteutettaessa nimittäin törmättiin useisiin pohjana toimineen Theron-aktorikirjaston rajoituksiin, jotka vaikeuttavat pelimoottorin kehittämistä ja saattavat aiheuttaa todellisessa moottorissa vakavia ongelmia – esimerkiksi Stage-moottorin ruudunpäivitysvaiheiden laskennan päättymisen tunnistamisessa käytetty ratkaisu on melko raskas ja saattaa virhetilanteessa aiheuttaa koko moottorin pysähtymisen, skriptikielillä ohjelmoituja pelioliokomponentteja olisi hankala toteuttaa viestityyppien käännösaikaisen määrittelyn vuoksi ja aktorikoodin kirjoittaminen on tavallista yksisäikeistä tai lukkopohjaista koodia työläämpää. Mikään näistä ei sinänsä ole aktorimallin vaan Stage-moottorin pohjaksi valitun aktorijärjestelmän syytä, joten osa havaituista ongelmista voitaisiin mahdollisesti korjata yksinkertaisesti käyttämällä jotakin muuta saatavilla olevaa kirjastoa. Todennäköisesti nykypeleihin soveltuvan moottorin toteuttaminen vaatisi kuitenkin koko aktorijärjestelmän räätälöimistä juuri pelimoottorin tarpeisiin, sillä esimerkiksi työntekijäsäikeiden tilan seuraaminen ei ole ominaisuus, jota aktoripohjaiset ohjelmat yleensä tarvitsevat.

Lähteet

AnJ09 Andrews, J., Designing the framework of a parallel game engine. Intel Corporation, 2009. http://www.design1online.com/downloads/whitepapers/parallelgameengine.pdf [27.4.2014]

AtD13 Atkins, D., Modern Concurrency Techniques: an Exploration. Maisterintutkielma, Victoria University of Wellington, 2013. http://hdl.handle.net/10063/3068 [15.1.2016]

BAM06 Ben-Ari, M., Principles of Concurrent and Distributed Programming, Second Edition. Addison-Wesley, 2006.

BiS02 Bilas, S., A data-driven game object system. Game Developers Conference Proceedings, 2002. http://gamedevs.org/uploads/data-driven-game-object-system.pdf [21.6.2014]

CAF16 CAF – C++ Actor Framework. http://actor-framework.org/ [1.1.2016]

DAR15 Dawes, B., Abrahams, D. ja Rivera, R., Boost C++ libraries. http://www.boost.org/ [12.7.2015]

DaW11 Damon, W., Multithreaded Game Programming and Hyper-Threading Technology. Intel Corporation, 2011. https://software.intel.com/en-us/articles/multithreaded-game-programming-and-hyper-threading-technology [6.5.2014]

DeM09 DeLoura, M., The Engine Survey: General results. Gamasutra, 2009. http://www.gamasutra.com/blogs/MarkDeLoura/20090302/581/The_Engine_Survey_General_results.php [15.5.20014]

GLE15 GLEW, GLEW: The OpenGL Extension Wrangler Library. http://glew.sourceforge.net/ [12.7.2015]

GLF15 GLFW, GLFW – An OpenGL library. http://www.glfw.org/ [12.7.2015]

GrJ09 Gregory, J., Game Engine Architecture. CRC Press, 2009.

GTr15 G-Truc Creation, OpenGL Mathematics. http://glm.g-truc.net/0.9.6/in-

dex.html [12.7.1015]

HBS73 Hewitt, C., Bishop, P. ja Steiger, R., A universal modular ACTOR formalism for artificial intelligence. IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence, 1973, sivut 235-245.

ISO12 ISO/IEC, Working Draft, Standard for Programming Language C++. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf [16.3.2016]

JoM08 Joselli, M. et al., A new physics engine with automatic process distribution between cpu-gpu. Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, 2008, sivut 149-156. http://dl.acm.org/citation.cfm?id=1401871 [9.4.2014]

Khr15 Khronos Group, OpenGL – The Industry Standard for High Performance Graphics. https://www.opengl.org/ [12.7.2015]

KuR13 Kumar, R., Efficient execution of fine-grained actors on multicore processors. Väitöskirja, University of Illinois at Urbana-Champaign, 2013. http://hdl.handle.net/2142/44758 [15.1.2016]

LaA10 Lake, A., Game Programming Gems 8. Cencage Learning, 2010.

LaG05 Lake, A. ja Gabb, H., Threading 3D Game Engine Basics. Gamasutra, 2005. http://www.gamasutra.com/view/feature/2463/threading_3d_game_engine_basics.php [28.10.2015]

MaA14 Mason, A., Theron documentation. http://www.theron-library.com/index.php?t=page&p=documentation [12.4.2014]

MaD02 Marr, D., et al, Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal Q, 2002.http://www.cs.sfu.ca/~fedorova/Teaching/CMPT886/Spring2007/papers/hyper-threading.pdf [26.1.2016]

McC15 McCaffrey, C., Building the Halo 4 Services with Orleans. QCon, 2015. http://www.infoq.com/presentations/halo-4-orleans [15.1.2016]

MiI07 Millington, I., Game Physics Engine Development. Morgan Kaufmann Pub-

lishers, 2007.

OSV08 Odersky, M., Spoon, L. ja Venners, B. Programming in Scala, luku 32: Actors and Concurrency. Artima Press, 2008, sivut 691-726.

PaE09 Passos, E., et al., Smart composition of game objects using dependency injection. Computers in Entertainment (CIE) - SPECIAL ISSUE: Games, Volume 7 Issue 4, 2009. http://dl.acm.org/citation.cfm?id=1658872

[21.6.2014]

PeT12 Petricek, T., An Introduction to F# Agents. DeveloperFusion, 2012. http://www.developerfusion.com/article/139804/an-introduction-to-f-agents/ [15.10.2014]

ReD14 Reisinger, D., Activision pumping $500 million into Bungie's 'Destiny'. CNET, 2014. http://www.cnet.com/news/activision-to-pump-500-million-into-bungies-destiny/ [8.5.2014]

TBN06 Tulip, J., Bekkema, J. ja Nesbitt, K., Multi-threaded game engine design. Proceedings of the 3rd Australasian conference on Interactive entertainment (2006), sivut 9-14. http://dl.acm.org/citation.cfm?id=1231896 [11.2.2014]

Uni15 Unity Technologies, Unity – Manual: GameObjects. http://docs.unity3d.com/Manual/GameObjects.html[10.1.2015]

VeR09 Vermeersch, R., Concurrency in Erlang & Scala: The Actor Model. 2009. https://savanne.be/articles/concurrency-in-erlang-scala/ [15.10.2014]

ZaM07 Zamith, M. P. D. M. et al., Parallel processing between GPU and CPU: Concepts in a game architecture. Computer Graphics, Imaging and Visualisation, 2007, sivut 115-120. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4293658 [18.3.2014]

Liite 1. Kaikille Stage-pelimoottoreille yhteiset moduulit

Tämä liite sisältää kaikille Stage-pelimoottoritoteutuksille (Stage, Stage Control ja Stage 11) yhteisten pelimoottorimoduulien ohjelmakoodin.

Moduuli Stage_common_graphics:

Camera.h:

```
#ifndef CAMERA_H
#define CAMERA_H

#include "stdafx.h"

namespace stage_common{
	/**	Luokka, joka sisältää grafiikkamoottorin kameran mallintamiseen
	vaadittavat tiedot.*/
	class Camera{
	public:
		/** Luo kameran oletusasetuksilla (45.0 asteen fov, kuvasuhde 4:3,
		piirtoetäisyys 0.1-100, suunnattu origoon)*/
		Camera();
		/** Luo kameran luojan määrittelemillä asetuksilla
		@param initialProjection	Kameran projektiomatriisi (fov,
		kuvasuhde, piirtoetäisyys) simulaation alussa
		@param initialView		Kameran näkymämatriisi (mihin
		kamera on suunnattu) simulaation alussa
		*/
		Camera(glm::mat4& initialProjection, glm::mat4& initialView);
		/** Palauttaa kameran nykyisen projektiomatriisin
		@returns	Kameran projektiomatriisi
		*/
		glm::mat4 getProjectionMatrix() const { return projection; }
		/** Palauttaa kameran nykyisen näkymämatriisin
		@returns	Kameran näkymämatriisi
		*/
		glm::mat4 getViewMatrix() const { return view; }
		/** Asettaa kameralle uuden projektiomatriisin
		@param newProjection Uusi projektiomatriisi
		*/
		void setProjectionMatrix(const glm::mat4& newProjection);
		/** Asettaa kameralle uuden näkymämatriisin
		@param newView	Uusi näkymämatriisi
		*/
		void setViewMatrix(const glm::mat4& newView);
	private:
		/** Kameran projektiomatriisi (fov, kuvasuhde, piirtoetäisyys)*/
		glm::mat4 projection;
		/** Kameran näkymämatriisi (mihin kamera on suunnattu)*/
		glm::mat4 view;
	};
}
#endif
```

Camera.cpp:

```
#include "stdafx.h"
#include "Camera.h"
#include <glm\gtc\matrix_transform.hpp>

using namespace stage_common;

Camera::Camera(){
	//45 asteen fov, kuvasuhde 4:3, piirtoetäisyys 0.1-100
	projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);
	//katsotaan pisteestä 1,1,1 origoon
	view = glm::lookAt(
		glm::vec3(1, 1, 1),
		glm::vec3(0, 0, 0),
		glm::vec3(0, 1, 0)
	);
}
Camera::Camera(glm::mat4& initialProjection, glm::mat4& initialView):
	projection(initialProjection), view(initialView){
}
void Camera::setProjectionMatrix(const glm::mat4& newProjection){
	projection = newProjection;
}
void Camera::setViewMatrix(const glm::mat4& newView){
	view = newView;
}
```

GraphicsController.h:

```
#ifndef GRAPHICSCONTROLLLER_H
#define GRAPHICSCONTROLLLER_H

#include "stdafx.h"
#include <string>
#include <vector>
#include "Model.h"
#include "Camera.h"

namespace stage_common{
	class Input;
	/** Luokka, joka toimii rajapintana pelimoottorin ja OpenGL:n välillä.
	Käynnistää OpenGL:n ja grafiikka-apukirjastot ja siirtää dataa moottorilta
	näytönohjaimelle.*/
	class GraphicsController{
	public:
		/** Käynnistää pelimoottorin grafiikkakomponentin ja luo uuden
		ikkunan.
		@param windowName	Ikkunan nimi
		@param xres		Ikkunan vaakaresoluutio
		@param yres		Ikkunan pystyresoluutio
		*/
		GraphicsController(std::string& windowName, int xres, int yres);
		/** Sammuttaa pelimoottorin grafiikkakomponentin, sulkee ikkunan ja
		sammuttaa grafiikka-apukirjastot.*/
		~GraphicsController();
		/** Määrittää 3D-mallin piirrettäväksi seuraavassa
		ruudunpäivityksessä.
		@param model  Osoitin piirrettävään 3D-malliin
		@param position	Matriisi, joka ilmaisee mihin, miten päin ja
```

```
		missä mittakaavassa malli piirretään
		*/
		void queue(const Model* model, const glm::mat4& position){
			//Sijoitettu headeriin oudon linkkaajabugin vuoksi
			if (drawCount >= models.size()){
				//Varataan tila uudelle mallille, jos sitä ei
				//ennestään ole
				models.push_back(model);
				positions.push_back(position);
			}
			else {
				//Jos mallille on tilaa (jossain edellisessä ruudussa
				//on piirretty ainakin yhtä monta mallia), käytetään
				//hyväksi olemassaolevaa tilanvarausta.
				models[drawCount] = model;
				positions[drawCount] = position;
			}
			drawCount++;
		}
		/** Piirtää ruudulle kaikki ne 3D-mallit, jotka on queue-metodilla
		tilattu pirrettäväksi edellisen draw-kutsun jälkeen.
		@param cam	Viite kameraolioon, jonka perspektiivistä mallit
				piirretään
		*/
		void draw(const Camera& cam);
		/** Kertoo, onko käyttäjä pyytänyt ohjelmaa pysähtymään.
		@returns	True, jos ohjelman suoritus tulisi pysäyttää
		*/
		bool shouldStop(){ return stopLoop; }
		/** Hakee globaalin GraphicsController-singletonin, jonka kautta
		kaikki piirtokutsut tulisi reitittää
		@returns	Osoitin GraphicsController-singletoniin
		*/
		static GraphicsController* getGlobalController();
	private:
		/** Osoitin globaaliin GraphicsController-singletoniin*/
		static GraphicsController* globalController;
		/** Osoitin olioon, joka pitää kirjaa nykyisen ruudunpäivityksen
		aikana luetuista syötteistä*/
		Input* input;
		/** Osoitin pelin ikkuna-olioon	*/
		GLFWwindow* window;
		/** True, jos ohjelman suoritus tulisi pysäyttää*/
		bool stopLoop = false;
		/** Lista seuraavan ruudunpäivityksen aikana piirrettävistä 3D-
		malleista*/
		std::vector<const Model*> models;
		/** Lista sijainneista, joihin seuraavan ruudunpäivityksen aikana
		piirretään 3D-malli*/
		std::vector<const glm::mat4> positions;
		/** Seuraavan ruudunpäivityksen aikana piirrettävien 3D-mallien
		lukumäärä*/
		unsigned int drawCount = 0;
	};
}
#endif
```

GraphicsController.cpp:

```
#include "stdafx.h"
#include "GraphicsController.h"
```

```
#include <iostream>
#include <Input.h>

using namespace stage_common;

GraphicsController* GraphicsController::globalController = nullptr;

GraphicsController::GraphicsController(std::string& windowName, int xres, int
yres){
	// Ei sallita enempää kuin 1 GraphicsController
	if (globalController != nullptr){
		std::cout << "global graphics controller already set\n";
		abort();
	}
	globalController = this;
	// Käynnistetään GLFW
	if (!glfwInit()){
		std::cout << "failed to init glfw\n";
		abort();
	}
	glfwWindowHint(GLFW_CLIENT_API, GLFW_OPENGL_API);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_COMPAT_PROFILE);
	//Luodaan ikkuna
	window = glfwCreateWindow(xres, yres, windowName.c_str(), NULL, NULL);
	if (!window){
		std::cout << "failed to create window\n";
		glfwTerminate();
		abort();
	}
	//Luodaan käyttäjän syötteet lukeva olio
	input = new Input(window);
	//Asetetaan OpenGL-operaatiot tehtäväksi tässä ikkunassa
	glfwMakeContextCurrent(window);
	//vsync
	glfwSwapInterval(1);
	//Käynnistetään GLEW
	if (glewInit()){
		std::cout << "failed to init glew\n";
		glfwDestroyWindow(window);
		glfwTerminate();
		abort();
	}
	//backface culling
	glEnable(GL_CULL_FACE);
	//Täytetään tyhjät alueet sinisellä
	glClearColor(0.0f, 0.0f, 0.4f, 0.0f);
	//Ei piirretä takana olevia polygoneja eteen
	glEnable(GL_DEPTH_TEST);
	glDepthFunc(GL_LESS);
}
GraphicsController::~GraphicsController(){
	globalController = nullptr;
	delete input;
	glfwDestroyWindow(window);
	glfwTerminate();
}
GraphicsController* GraphicsController::getGlobalController(){
	return globalController;
}
```

```
void GraphicsController::draw(const Camera& cam){
	//Tyhjennetään grafiikkapuskuri
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	//piirretään kaikki tilatut mallit
	for (unsigned int i = 0; i < drawCount; i++){
		models[i]->draw(cam, positions[i]);
	}
	//resetoidaan piirrettävien mallien määrä seuraavaa ruutua varten
	drawCount = 0;

	glfwSwapBuffers(window);
	glfwPollEvents();
	if (glfwWindowShouldClose(window)) stopLoop = true;
}
```

Model.h:

```
#ifndef MODEL_H
#define MODEL_H

#include "stdafx.h"
#include "Camera.h"
#include <vector>

namespace stage_common{
	class Shader;
	/** Luokka, joka sisältää tiedot 3D-mallin piirtämistä varten.
	Jokaista eri 3D-mallia kohden tulisi olla tasan yksi Model-olio.*/
	class Model{
	public:
		/** Luo uuden 3D-mallin pelimoottorin muistiin.
		@param vertexData	Mallin verteksit listana kolmen koordinaatin
					vektoreita.
		@param colorData	Mallin verteksien värit listana kolmen
					väriarvon vektoreita.
		@param shader		Tämän mallin piirtämiseen kaytettävä
					sävytinolio.
		*/
		Model(std::vector<glm::vec3>& vertexData, std::vector<glm::vec3>&
			colorData, Shader* shader);
		/** Palauttaa sijainnin, jonne OpenGL on ladannut tämän mallin
		verteksit
		@returns	Mallin verteksien sijaintia kuvaava GLuint
		*/
		GLuint getVertices() const { return vertices; }
		/** Palauttaa sijainnin, jonne OpenGL on ladannut tämän mallin
		värit
		@returns	Mallin värien sijaintia kuvaava GLuint
		*/
		GLuint getColors() const { return colors; }
		/** Kertoo, miten monta verteksiä tässä mallissa on
		@returns	Verteksien lukumäärä
		*/
		unsigned int getVertexCount() const { return vertexCount; }
		/** Piirtää tämän 3D-mallin ruudulle
		@param cam		Kameraolio, jonka perspektiivistä malli
					piirretään
		@param modelMatrix	Mallin sijaintia, kiertoa ja skaalausta
					kuvaava matriisi
		*/
		void draw(const Camera& cam, glm::mat4& modelMatrix) const;
```

```
	private:
		/** Mallin verteksien sijaintia kuvaava GLuint*/
		GLuint vertices;
		/** Mallin värien sijaintia kuvaava GLuint*/
		GLuint colors;
		/** Osoitin sävytinolioon, jonka sävytinohjelmalla tämä malli
		piirretään*/
		Shader* shader;
		/** Mallin verteksien lukumäärä*/
		unsigned int vertexCount;
	};
}
#endif
```

Model.cpp:

```
#include "stdafx.h"
#include "Model.h"
#include "Shader.h"
#include <iostream>

using namespace stage_common;

Model::Model(std::vector<glm::vec3>& vertexData, std::vector<glm::vec3>&
	colorData, Shader* shader): vertexCount(vertexData.size()), shader(shader)
{
	//Ladataan OpenGL:n muistiin 3D-mallin verteksit
	glGenBuffers(1, &vertices);
	glBindBuffer(GL_ARRAY_BUFFER, vertices);
	glBufferData(GL_ARRAY_BUFFER, vertexData.size() * sizeof(glm::vec3),
		&vertexData[0], GL_STATIC_DRAW);
	//Ladataan OpenGL:n muistiin 3D-mallin verteksien värit
	glGenBuffers(1, &colors);
	glBindBuffer(GL_ARRAY_BUFFER, colors);
	glBufferData(GL_ARRAY_BUFFER, colorData.size() * sizeof(glm::vec3),
		&colorData[0], GL_STATIC_DRAW);
}
void Model::draw(const Camera& cam, glm::mat4& modelMatrix) const{
	shader->draw(*this, cam, modelMatrix);
}
```

Shader.h:

```
#ifndef SHADER_H
#define SHADER_H

#include "stdafx.h"
#include "Model.h"
#include "Camera.h"

namespace stage_common{
	/** Abstrakti luokka, joka mallintaa OpenGL-sävytinohjelman
	*/
	class Shader{
	public:
		/** Lataa muistiin uuden sävytinohjelman
		@param vertexShader		Ohjelman verteksisävytin
		@param fragmentShader	Ohjelman pikselisävytin
		*/
		Shader(const char* vertexShader, const char* fragmentShader);
		/** Abstrakti metodi, joka piirtää mallin ruudulle tämän
		sävytinohjelman avulla
```

```
		@param model		Piirrettävä 3D-malli
		@param cam		Kamera, jonka kuvakulmasta malli piirretään
		@param modelMatrix	Matriisi, joka ilmaisee mihin malli piirretään
		*/
		virtual void draw(const Model& model, const Camera& cam, glm::mat4&
			modelMatrix) = 0;
	protected:
		/** Ilmaisee sävytinohjelman osoitteen OpenGL:n muistissa*/
		GLuint program;
	};
}
#endif
```

Shader.cpp:

```
#include "stdafx.h"
#include "Shader.h"
#include <string>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace stage_common;

/** Apufunktio, joka lataa ja kääntää sävytinohjelman
Julkaistu alun perin osoitteessa http://www.opengl-tutorial.org/download/ WTFPL2-
lisenssin alaisuudessa (http://www.wtfpl.net/)
@param vertex_file_path		Osoite verteksisävyttimen tiedostoon
@param fragment_file_path	Osoite pikselisävyttimen tiedostoon
@returns			GLuint, joka ilmaisee minne OpenGL
latasi käännetyn sävytinohjelman
*/
GLuint LoadShaders(const char * vertex_file_path, const char *
fragment_file_path){
	// Create the shaders
	GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
	GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
	// Read the Vertex Shader code from the file
	std::string VertexShaderCode;
	std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
	if (VertexShaderStream.is_open())
	{
		std::string Line = "";
		while (getline(VertexShaderStream, Line)){
			VertexShaderCode += "\n" + Line;
		}
		VertexShaderStream.close();
	}
	// Read the Fragment Shader code from the file
	std::string FragmentShaderCode;
	std::ifstream FragmentShaderStream(fragment_file_path, std::ios::in);
	if (FragmentShaderStream.is_open()){
		std::string Line = "";
		while (getline(FragmentShaderStream, Line))
			FragmentShaderCode += "\n" + Line;
		FragmentShaderStream.close();
	}

	GLint Result = GL_FALSE;
	int InfoLogLength;
	// Compile Vertex Shader
```

```
        printf("Compiling shader : %s\n", vertex_file_path);
        char const * VertexSourcePointer = VertexShaderCode.c_str();
        glShaderSource(VertexShaderID, 1, &VertexSourcePointer, NULL);
        glCompileShader(VertexShaderID);
        // Check Vertex Shader
        glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
        glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
        std::vector<char> VertexShaderErrorMessage(std::max(InfoLogLength,
               int(1)));
        glGetShaderInfoLog(VertexShaderID, InfoLogLength, NULL,
               &VertexShaderErrorMessage.at(0));
        fprintf(stdout, "%s\n", &VertexShaderErrorMessage.at(0));
        // Compile Fragment Shader
        printf("Compiling shader : %s\n", fragment_file_path);
        char const * FragmentSourcePointer = FragmentShaderCode.c_str();
        glShaderSource(FragmentShaderID, 1, &FragmentSourcePointer, NULL);
        glCompileShader(FragmentShaderID);
        // Check Fragment Shader
        glGetShaderiv(FragmentShaderID, GL_COMPILE_STATUS, &Result);
        glGetShaderiv(FragmentShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
        std::vector<char> FragmentShaderErrorMessage(std::max(InfoLogLength,
               int(1)));
        glGetShaderInfoLog(FragmentShaderID, InfoLogLength, NULL,
               &FragmentShaderErrorMessage[0]);
        fprintf(stdout, "%s\n", &FragmentShaderErrorMessage[0]);
        // Link the program
        fprintf(stdout, "Linking program\n");
        GLuint ProgramID = glCreateProgram();
        glAttachShader(ProgramID, VertexShaderID);
        glAttachShader(ProgramID, FragmentShaderID);
        glLinkProgram(ProgramID);
        // Check the program
        glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
        glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &InfoLogLength);
        std::vector<char> ProgramErrorMessage(std::max(InfoLogLength, int(1)));
        glGetProgramInfoLog(ProgramID, InfoLogLength, NULL,
               &ProgramErrorMessage[0]);
        fprintf(stdout, "%s\n", &ProgramErrorMessage[0]);

        glDeleteShader(VertexShaderID);
        glDeleteShader(FragmentShaderID);

        return ProgramID;
}
stage_common::Shader::Shader(const char* vertexShader, const char*
        fragmentShader){
        program = LoadShaders(vertexShader, fragmentShader);
}
```

SimpleShader.h:

```
#ifndef SIMPLESHADER_H
#define SIMPLESHADER_H

#include "stdafx.h"
#include "Shader.h"

namespace stage_common{
        /** Yksinkertainen sävytinohjelma, joka piirtää ja värittää
        tekstuurittoman 3D-mallin
        */
```

```
	class SimpleShader : public Shader {
	public:
		/** Lataa sävytinohjelman*/
		SimpleShader();
		/** Metodi, joka piirtää mallin ruudulle tämän sävytinohjelman
		avulla
		@param model		Piirrettävä 3D-malli
		@param cam			Kamera, jonka kuvakulmasta malli
		piirretään
		@param modelMatrix	Matriisi, joka ilmaisee mihin malli piirretään
		*/
		void draw(const Model& model, const Camera& cam, glm::mat4&
			modelMatrix);
	private:
		/** Sävytinohjelman käyttämän muuttujan osoite*/
		GLuint matrixID;
	};
}
#endif
```

SimpleShader.cpp:

```
#include "stdafx.h"
#include "SimpleShader.h"

#include <glm\gtc\matrix_transform.hpp>
#include <iostream>

using namespace stage_common;

SimpleShader::SimpleShader() : Shader("simplevertexshader.glsl",
	"simplefragmentshader.glsl"){
	matrixID = glGetUniformLocation(program, "MVP");
}
void SimpleShader::draw(const Model& model, const Camera& cam, glm::mat4&
	modelMatrix){
	//Lasketaan mallin sijainti kameraan nähden
	glm::mat4 MVP = cam.getProjectionMatrix() * cam.getViewMatrix() *
		modelMatrix;
	glUseProgram(program);
	//Asetetaan mallin sijainti sävytinohjelman muuttujaan
	glUniformMatrix4fv(matrixID, 1, GL_FALSE, &MVP[0][0]);
	//Annetaan verteksit sävyttimelle
	glEnableVertexAttribArray(0);
	glBindBuffer(GL_ARRAY_BUFFER, model.getVertices());
	glVertexAttribPointer(
		0,			// Sävytinohjelman parametri 0
		3,			// alkion koko
		GL_FLOAT,		// tyyppi
		GL_FALSE,		// normalisointi
		0,			// alkioiden välinen etäisyys
		(void*)0		// ensimmäisen alkion offset
	);
	//Annetaan värit sävyttimelle
	glEnableVertexAttribArray(1);
	glBindBuffer(GL_ARRAY_BUFFER, model.getColors());
	glVertexAttribPointer(
		1,			// Sävytinohjelman parametri 1
		3,
		GL_FLOAT,
		GL_FALSE,
```

```
			0,
			(void*)0
		);
		//Piirretään polygonit
		glDrawArrays(GL_TRIANGLES, 0, model.getVertexCount());
		glDisableVertexAttribArray(0);
		glDisableVertexAttribArray(1);
}
```

stdafx.h:

```
#pragma once
#define GLEW_STATIC

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from Windows
headers

#include <GL\glew.h>
#include <GLFW\glfw3.h>
#include <glm\glm.hpp>
```

Moduuli Stage_common_physics:

AABBCollider.h:

```
#pragma once
#ifndef AABBCOLLIDER_H
#define AABBCOLLIDER_H

#include "stdafx.h"
#include "Collider.h"
#include "Collisions.h"

namespace stage_common{
	class SphereCollider;
	/** Luokka, joka mallintaa pelimaailman akselien suuntaisen 3D-laatikon
	(Axis-aligned bounding box)
	törmäysten tunnistamista varten
	*/
	class AABBCollider : public Collider{
	public:
		/** Laatikon koko, eli se, miten kauas keskipisteestään laatikko
		ylettyy jokaista 3D-maailman akselia pitkin*/
		glm::vec3 size;
		/** Luo uuden AABB-törmäyshahmon
		@param size		Laatikon kokoa kuvaava vektori, eli miten
					kauas keskipisteestä laatikko ylettyy eri
					akseleita pitkin
		@param center Laatikon keskipisteen sijainti 3D-avaruudessa
		*/
		AABBCollider(glm::vec3 size, glm::vec3 center) : size(size),
			Collider(center){}
		/** Luo kopion AABB-törmäyshahmosta
		@param other  Kopioitava törmäyshahmo
		*/
		AABBCollider(const AABBCollider& other) : size(other.size),
			Collider(other.center){}
		/** Asettaa tämän törmäyshahmon tilan vastaamaan toista AABB-
```

```
törmäyshahmoa
@param other  Kopioitava törmäyshahmo
@returns            Viite tähän törmäyshahmoon
*/
AABBCollider& operator= (const AABBCollider& other){
	this->center = other.center;
	this->size = other.size;
	return *this;
}
/** Virtuaalikopiometodi, joka luo kopion tästä törmäyshahmosta
@returns      Osoitin luotuun kopioon. Huom: tuhottava
manuaalisesti deletellä
*/
Collider* copy() const{
	Collider* ret = new AABBCollider(*this);
	return ret;
}
/** Tarkistaa, onko tämä törmäyshahmo törmännyt toiseen
@param other  Toinen törmäyshahmo
@param returns       Palauttaa True, jos törmäyshahmot ovat
törmänneet
*/
bool checkCollision(const Collider& other) const{
	return other.checkCollision(*this);
}
/** Tarkistaa, onko tämä törmäyshahmo törmännyt pallotörmäyshamoon
@param other         Pallotörmäyshahmo
@param returns       Palauttaa True, jos törmäyshahmot ovat
                     törmänneet
*/
bool checkCollision(const SphereCollider& other) const{
	return Collisions::collision_sphere_aabb(other, *this);
}
/** Tarkistaa, onko tämä törmäyshahmo törmännyt AABB-törmäyshahmoon
@param other         AABB-törmäyshahmo
@param returns       Palauttaa True, jos törmäyshahmot ovat
                     törmänneet
*/
bool checkCollision(const AABBCollider& other) const{
	return Collisions::collision_aabb_aabb(*this, other);
}
/** Laskee kahden törmäyshahmon törmäyskohdan normaalin
@param other  Toinen törmäyshahmo
@param v              Toisen törmäyshahmon nopeus ennen törmäystä
@param returns       Vektori, joka kuvaa törmäyshahmojen
                     törmäyspinnan normaalia
*/
glm::vec3 getCollisionNormal(const Collider& other, const
	glm::vec3& v) const{
	//Tallennetaan liikesuunta vektoriin, jotta tiedetään mitkä
	//laatikon reunat voivat osallistua törmäykseen
	glm::vec3 signs;
	for (int i = 0; i < 3; i++){
		signs[i] = (v[i] < 0 ? -1 : 1);
	}
	//Tämän törmäyshahmon kulma, joka on toisen törmäyshahmon
	//sisässä
	glm::vec3 thisCorner;
	//Toisen törmäyshahmon kulma, joka on tämän törmäyshahmon
	//sisässä
	glm::vec3 otherCorner;
```

```
                thisCorner.x = (signs.x < 0 ? maxX() : minX());
                thisCorner.y = (signs.y < 0 ? maxY() : minY());
                thisCorner.z = (signs.z < 0 ? maxZ() : minZ());
                otherCorner.x = (signs.x < 0 ? other.maxX() : other.minX())
                        - v.x;
                otherCorner.y = (signs.y < 0 ? other.maxY() : other.minY())
                        - v.y;
                otherCorner.z = (signs.z < 0 ? other.maxZ() : other.minZ())
                        - v.z;
                //Kulmien etäisyys toisistaan ennen törmäystä
                glm::vec3 distance = (thisCorner - otherCorner) * signs;
                glm::vec3 normal;
                //Törmäysnormaali on todennäköisimmin pisimmän
                //etäisyysakselin suuntainen
                if (distance.x > distance.y && distance.x > distance.z)
                        normal = glm::vec3(-1, 0, 0);
                else if (distance.y > distance.z) normal = glm::vec3(0, -1,
                        0);
                else normal = glm::vec3(0, 0, -1);
                return (normal * signs);
        }
        /** Palauttaa tämän törmäyshahmon pienimmän x-koordinaatin
        @returns      Tämän törmäyshahmon pienin x-koordinaatti
        */
        virtual float minX() const{
                return center.x - size.x;
        }
        /** Palauttaa tämän törmäyshahmon pienimmän y-koordinaatin
        @returns      Tämän törmäyshahmon pienin y-koordinaatti
        */
        virtual float minY() const{
                return center.y - size.y;
        }
        /** Palauttaa tämän törmäyshahmon pienimmän z-koordinaatin
        @returns      Tämän törmäyshahmon pienin z-koordinaatti
        */
        virtual float minZ() const{
                return center.z - size.z;
        }
        /** Palauttaa tämän törmäyshahmon suurimman x-koordinaatin
        @returns      Tämän törmäyshahmon suurin x-koordinaatti
        */
        virtual float maxX() const{
                return center.x + size.x;
        }
        /** Palauttaa tämän törmäyshahmon suurimman y-koordinaatin
        @returns      Tämän törmäyshahmon suurin y-koordinaatti
        */
        virtual float maxY() const{
                return center.y + size.y;
        }
        /** Palauttaa tämän törmäyshahmon suurimman z-koordinaatin
        @returns      Tämän törmäyshahmon suurin z-koordinaatti
        */
        virtual float maxZ() const{
                return center.z + size.z;
        }
};
}
#endif
```

Collider.h:

```
#ifndef COLLIDER_H
#define COLLIDER_H

#include "stdafx.h"
#include <glm\glm.hpp>

namespace stage_common{
	class SphereCollider;
	class AABBCollider;
	/** Abstrakti luokka, joka mallintaa törmäyshahmon törmäysten
	tunnistamista varten
	*/
	class Collider{
	public:
		/** Tämän törmäyshahmon keskipiste 3D-avaruudessa
		*/
		glm::vec3 center;
		/** Luo uuden törmäyshahmon
		@param center Törmäyshahmon keskipiste 3D-avaruudessa
		*/
		Collider(glm::vec3 center) : center(center){}
		/** Tarkistaa, onko tämä törmäyshahmo törmännyt toiseen
		@param other  Toinen törmäyshahmo
		@param returns	Palauttaa True, jos törmäyshahmot ovat
		törmänneet
		*/
		virtual bool checkCollision(const Collider& other) const = 0;
		/** Tarkistaa, onko tämä törmäyshahmo törmännyt pallotörmäyshamoon
		@param other	Pallotörmäyshahmo
		@param returns	Palauttaa True, jos törmäyshahmot ovat
		törmänneet
		*/
		virtual bool checkCollision(const SphereCollider& other) const = 0;
		/** Tarkistaa, onko tämä törmäyshahmo törmännyt AABB-törmäyshahmoon
		@param other	AABB-törmäyshahmo
		@param returns	Palauttaa True, jos törmäyshahmot ovat
		törmänneet
		*/
		virtual bool checkCollision(const AABBCollider& other) const = 0;
		/** Virtuaalikopiometodi, joka luo kopion tästä törmäyshahmosta
		@returns	Osoitin luotuun kopioon. Huom: tuhottava
				manuaalisesti deletellä
		*/
		virtual Collider* copy() const = 0;
		/** Laskee kahden törmäyshahmon törmäyskohdan normaalin
		@param other  Toinen törmäyshahmo
		@param v		Toisen törmäyshahmon nopeus ennen törmäystä
		@param returns	Vektori, joka kuvaa törmäyshahmojen
				törmäysnormaalia
		*/
		virtual glm::vec3 getCollisionNormal(const Collider& other, const
			glm::vec3& v) const = 0;
		/** Palauttaa tämän törmäyshahmon pienimmän x-koordinaatin
		@returns	Tämän törmäyshahmon pienin x-koordinaatti
		*/
		virtual float minX() const = 0;
		/** Palauttaa tämän törmäyshahmon pienimmän y-koordinaatin
		@returns	Tämän törmäyshahmon pienin y-koordinaatti
```

```
		*/
		virtual float minY() const = 0;
		/** Palauttaa tämän törmäyshahmon pienimmän z-koordinaatin
		@returns	Tämän törmäyshahmon pienin z-koordinaatti
		*/
		virtual float minZ() const = 0;
		/** Palauttaa tämän törmäyshahmon suurimman x-koordinaatin
		@returns	Tämän törmäyshahmon suurin x-koordinaatti
		*/
		virtual float maxX() const = 0;
		/** Palauttaa tämän törmäyshahmon suurimman y-koordinaatin
		@returns	Tämän törmäyshahmon suurin y-koordinaatti
		*/
		virtual float maxY() const = 0;
		/** Palauttaa tämän törmäyshahmon suurimman z-koordinaatin
		@returns	Tämän törmäyshahmon suurin z-koordinaatti
		*/
		virtual float maxZ() const = 0;
	};
}
#endif
```

Collisions.h:

```
#ifndef COLLISIONS_H
#define COLLISIONS_H

#include "stdafx.h"
#include <glm\glm.hpp>

namespace stage_common{
	class SphereCollider;
	class AABBCollider;
	class Collider;
	/** Apuluokka, joka sisältää staattisia funktioita fysiikkamoottorin
		laskelmien suorittamiseksi
	*/
	class Collisions{
	public:
		/** Laskee, onko pallotörmäyshahmo törmännyt toiseen
		@param a	Pallotörmäyshahmo
		@param b	Toinen pallotörmäyshahmo
		@returns	True, jos törmäyshahmot ovat törmänneet
		*/
		static bool collision_sphere_sphere(const SphereCollider& a, const
			SphereCollider& b);
		/** Laskee, onko pallotörmäyshahmo törmännyt AABB-törmäyshahmoon
		@param a	Pallotörmäyshahmo
		@param b	AABB-törmäyshahmo
		@returns	True, jos törmäyshahmot ovat törmänneet
		*/
		static bool collision_sphere_aabb(const SphereCollider& a, const
			AABBCollider& b);
		/** Laskee, onko AABB-törmäyshahmo törmännyt toiseen
		@param a	AABB-törmäyshahmo
		@param b	Toinen AABB-törmäyshahmo
		@returns	True, jos törmäyshahmot ovat törmänneet
		*/
		static bool collision_aabb_aabb(const AABBCollider& a, const
			AABBCollider& b);
		/** Siirtää törmäyksen jälkeen törmäyshahmoa taaksepäin, kunnes se
```

```
		ei enää ole törmäyksessä toisen kanssa
		@param mover  Taaksepäin siirtyvä törmäyshahmo
		@param velocity	Siirtyvän törmäyshahmon nopeus
		@param hit		Törmäyshahmo, johon mover törmäsi
		*/
		static void backOff(Collider& mover, glm::vec3& velocity, const
			Collider& hit);
		/** Laskee törmänneille kappaleille uudet nopeudet
		@param v1	Ensimmäisen kappaleen nopeus
		@param m1	Ensimmäisen kappaleen massa
		@param v2	Toisen kappaleen nopeus
		@param m2	Toisen kappaleen massa
		*/
		static void collisionVelocityChange(glm::vec3& v1, float m1,
			glm::vec3& v2, float m2);
		/** "Kimmottaa" kappaleen pinnasta, eli laksee uuden nopeuden, kun
		kappale törmää liikkumattomaan pintaan
		@param v		Kappaleen nopeus
		@param normal Törmäyspinnan normaali
		@returns		Kappaleen nopeus törmäyksen jälkeen
		*/
		static glm::vec3 reflect(glm::vec3& v, glm::vec3 normal);
	};
}
#endif
```

Collisions.cpp:

```
#include "stdafx.h"
#include "Collider.h"
#include "Collisions.h"
#include "SphereCollider.h"
#include "AABBCollider.h"
#include <iostream>

using namespace stage_common;

bool Collisions::collision_sphere_sphere(const SphereCollider& a, const
	SphereCollider& b){
	return glm::length(a.center - b.center) < (a.radius + b.radius);
}
bool Collisions::collision_sphere_aabb(const SphereCollider& a, const
	AABBCollider& b){
	//Arvo's algorithm
	float squaredDistance = 0;
	float diff = 0;
	//iteroidaan x,y,z
	for (int i = 0; i < 3; i++){
		if (a.center[i] < (b.center[i] - b.size[i])){
			diff = a.center[i] - (b.center[i] - b.size[i]);
			squaredDistance += diff * diff;
		}
		if (a.center[i] > (b.center[i] + b.size[i])){
			diff = a.center[i] - (b.center[i] + b.size[i]);
			squaredDistance += diff * diff;
		}
	}
	return (squaredDistance <= a.radius * a.radius);
}
bool Collisions::collision_aabb_aabb(const AABBCollider& a, const AABBCollider&
	b){
```

```
	//Ovatko keskikohdat lähenpänä kuin laatikoiden koko?
	return
		glm::abs(a.center.x - b.center.x) < (a.size.x + b.size.x) &&
		glm::abs(a.center.y - b.center.y) < (a.size.y + b.size.y) &&
		glm::abs(a.center.z - b.center.z) < (a.size.z + b.size.z);
}
void Collisions::collisionVelocityChange(glm::vec3& v1, float m1, glm::vec3& v2,
	float m2){
	glm::vec3 res1 = (v1 * (m1 - m2) + 2 * m2 * v2) / (m1 + m2);
	glm::vec3 res2 = (v2 * (m2 - m1) + 2 * m1 * v1) / (m1 + m2);
	v1 = res1;
	v2 = res2;
}
void Collisions::backOff(Collider& mover, glm::vec3& velocity, const Collider&
	hit){
	while (mover.checkCollision(hit)){
		mover.center = mover.center - velocity;
	}
}
glm::vec3 Collisions::reflect(glm::vec3& v, glm::vec3 normal){
	return (-2 * (glm::dot(v, normal)) * normal + v);
}
```

SphereCollider.h:

```
#pragma once
#ifndef SPHERECOLLIDER_H
#define SPHERECOLLIDER_H

#include "stdafx.h"
#include "Collider.h"
#include "Collisions.h"

namespace stage_common{
	class AABBCollider;
	/** Luokka, joka mallintaa 3D-pallon törmäysten tunnistamista varten
	*/
	class SphereCollider : public Collider{
	public:
		/** Törmäyspallon säde
		*/
		float radius;
		/** Luo uuden pallotörmäyshahmon
		@param radius Pallon säde
		@param center Pallon keskipisteen sijainti 3D-avaruudessa
		*/
		SphereCollider(float radius, glm::vec3 center) : radius(radius),
			Collider(center){}
		/** Luo kopion pallotörmäyshahmosta
		@param other  Kopioitava törmäyshahmo
		*/
		SphereCollider(const SphereCollider& other) : radius(other.radius),
			Collider(other.center){}
		/** Asettaa tämän törmäyshahmon tilan vastaamaan toista
		pallotörmäyshahmoa
		@param other  Kopioitava törmäyshahmo
		@returns		Viite tähän törmäyshahmoon
		*/
		SphereCollider& operator= (const SphereCollider& other){
			this->center = other.center;
			this->radius = other.radius;
```

```
        return *this;
}
/** Virtuaalikopiometodi, joka luo kopion tästä törmäyshahmosta
@returns      Osoitin luotuun kopioon. Huom: tuhottava
              manuaalisesti deletellä
*/
Collider* copy() const{
        Collider* ret = new SphereCollider(*this);
        return ret;
}
/** Tarkistaa, onko tämä törmäyshahmo törmännyt toiseen
@param other  Toinen törmäyshahmo
@param returns        Palauttaa True, jos törmäyshahmot ovat
                      törmänneet
*/
bool checkCollision(const Collider& other) const{
        return other.checkCollision(*this);
}
/** Tarkistaa, onko tämä törmäyshahmo törmännyt pallotörmäyshamoon
@param other  Pallotörmäyshahmo
@param returns        Palauttaa True, jos törmäyshahmot ovat
                      törmänneet
*/
bool checkCollision(const SphereCollider& other) const{
        return Collisions::collision_sphere_sphere(*this, other);
}
/** Tarkistaa, onko tämä törmäyshahmo törmännyt AABB-törmäyshahmoon
@param other  AABB-törmäyshahmo
@param returns        Palauttaa True, jos törmäyshahmot ovat
                      törmänneet
*/
bool checkCollision(const AABBCollider& other) const{
        return Collisions::collision_sphere_aabb(*this, other);
}
/** Laskee kahden törmäyshahmon törmäyskohdan normaalin
@param other  Toinen törmäyshahmo
@param v              Toisen törmäyshahmon nopeus ennen törmäystä
@param returns        Vektori, joka kuvaa törmäyshahmojen
                      törmäysnormaalia
*/
glm::vec3 getCollisionNormal(const Collider& other, const
        glm::vec3& v) const{
        return glm::normalize(other.center - v - center);
}
/** Palauttaa tämän törmäyshahmon pienimmän x-koordinaatin
@returns      Tämän törmäyshahmon pienin x-koordinaatti
*/
virtual float minX() const{
        return center.x - radius;
}
/** Palauttaa tämän törmäyshahmon pienimmän y-koordinaatin
@returns      Tämän törmäyshahmon pienin x-koordinaatti
*/
virtual float minY() const{
        return center.y - radius;
}
/** Palauttaa tämän törmäyshahmon pienimmän z-koordinaatin
@returns      Tämän törmäyshahmon pienin x-koordinaatti
*/
virtual float minZ() const{
        return center.z - radius;
```

```
		}
		/** Palauttaa tämän törmäyshahmon suurimman x-koordinaatin
		@returns	Tämän törmäyshahmon suurin x-koordinaatti
		*/
		virtual float maxX() const{
			return center.x + radius;
		}
		/** Palauttaa tämän törmäyshahmon suurimman y-koordinaatin
		@returns	Tämän törmäyshahmon suurin y-koordinaatti
		*/
		virtual float maxY() const{
			return center.y + radius;
		}
		/** Palauttaa tämän törmäyshahmon suurimman z-koordinaatin
		@returns	Tämän törmäyshahmon suurin z-koordinaatti
		*/
		virtual float maxZ() const{
			return center.z + radius;
		}
	};
}
#endif
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN
```

Moduuli Stage_common_utilities:

Input.h:

```
#ifndef INPUT_H
#define INPUT_H

#include "stdafx.h"
#include <GLFW\glfw3.h>
#include <list>
#include <unordered_map>
#include <iostream>

namespace stage_common{
	/** Käyttäjän syötteitä lukeva luokka*/
	class Input{
		//Syötteiden lukemista varten tarvitaan osoitin peli-ikkunaan
		friend class GraphicsController;
	public:
		/** Hakee rekisteröityjen näppäimien nykytilan
		@param rm	Ilmaisee, palautetaanko hiiri ruudun keskelle
		*/
		void update(bool rm){
			//Haetaan kaikkien rekisteröityjen näppäinten tila
			for (unsigned int i = 0; i < keys.size(); i++){
				lastInputs[keys[i]] = (glfwGetKey(window, keys[i]) ==
					GLFW_PRESS);
			}
			//Haetaan hiiriosoittimen sijainti ruudulla
```

```
			glfwGetCursorPos(window, &cursorx, &cursory);
			glfwGetWindowSize(window, &xres, &yres);
			if (rm){
				//Palautetaan hiiri keskelle ruutua
				resetMouse();
			}
		}
		/** Pyytää input-oliota tarkkailemaan halutun näppäimen tilaa
		@param key	Tarkkailtavan näppäimen GLFW-tunnus
		*/
		void registerKey(int key){
			if (lastInputs.count(key) == 0){
				keys.push_back(key);
				lastInputs.emplace(key, false);
			}
		}
		/** Kysyy, oliko tietty näppäin pohjassa tämän ruudunpäivityksen
		alussa
		@param key	Näppäimen GLFW-tunnus
		@returns	true, jos näppäin oli pohjassa
		*/
		bool getKeyDown(int key){
			return lastInputs[key];
		}
		/** Hakee hiiriosoittimen x-koordinaatin suhteessa peli-ikkunaan
		@returns	Arvo väliltä 0 (ruudun vasen reuna) - 1 (ruudun oikea
				reuna)
		*/
		double getCursorX(){
			return cursorx / (double)xres;
		}
		/** Hakee hiiriosoittimen y-koordinaatin suhteessa peli-ikkunaan
		@returns	Arvo väliltä 0 (ruudun yläreuna) - 1 (ruudun
				alareuna)
		*/
		double getCursorY(){
			return cursory / (double)yres;
		}
		/** Hakee osoittimen globaaliin Input-singletoniin
		@returns	Globaalin Input-olion osoite
		*/
		static Input& getSingleton(){
			return *singleton;
		}
		/** Asettaa hiiren keskelle peli-ikkunaa
		*/
		void resetMouse(){
			glfwSetCursorPos(window, xres / 2, yres / 2);
		}
	private:
		/**Luo uuden Input-olion
		@param window Osoitin peli-ikkunaolioon
		*/
		Input(GLFWwindow* window) : window(window){
			singleton = this;
			//Piilotetaan hiiri, kun se on peli-ikkunan päällä
			glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_HIDDEN);
		}
		/** Tuhoaa Input-olion*/
		~Input(){
			if (singleton == this) singleton = nullptr;
```

```
		}
		/** Globaalin syötteidenhallintaolion osoite*/
		static Input* singleton;
		/** Peli-ikkunan osoite*/
		GLFWwindow* window;
		/** Lista niistä näppäimistä, joiden tilaa tarkkaillaan*/
		std::vector<int> keys;
		/** Lista tarkkailtavien näppäimien tiloista ruudunpäivityksen
		alussa*/
		std::unordered_map<int, bool> lastInputs;
		/** Hiiriosoittimen x- ja y-koordinaatit ruudunpäivityksen alussa*/
		double cursorx = 1, cursory = 1;
		/** Peli-ikkunan vaaka- ja pystyresoluutio*/
		int xres = 2, yres = 2;
	};
}
#endif
```

Input.cpp

```
#include "stdafx.h"
#include "Input.h"

using namespace stage_common;

Input* Input::singleton = nullptr;
```

Logger.h:

```
#ifndef LOGGER_H
#define LOGGER_H

#include "stdafx.h"
#include <iostream>
#include <string>

namespace stage_common{
	/**Luokka, joka tarjoaa yksinkertaisen lokitoiminnon.*/
	class Logger{
	public:
		/** Luo uuden lokiolion
		@param standard			Minne normaalit lokiviestit kirjoitetaan
		@param error		Minne virheilmoitukset kirjoitetaan
		*/
		Logger(std::ostream& standard, std::ostream& error) :
			stdLog(standard), errLog(error){
		}
		/** Kirjoittaa lokiin normaalin viestin
		@param msg	Lokiin kirjoitettava viesti
		*/
		void Log(std::string msg){
			stdLog << msg << std::endl;
		}
		/** Kirjoittaa lokiin virheilmoituksen
		@param msg	Lokiin kirjoitettava viesti
		*/
		void LogError(std::string msg){
			errLog << msg << std::endl;
		}
	private:
		/**Kanava, jonne kirjoitetaan normaalit lokiviestit*/
		std::ostream& stdLog;
```

```
		/**Kanava, jonne kirjoitetaan virheilmoitukset */
		std::ostream& errLog;
	};
}

#endif
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN
```

Timer.h:

```
#ifndef TIMER_H
#define TIMER_H

#include "stdafx.h"
#include <chrono>

typedef std::chrono::time_point<std::chrono::high_resolution_clock> clock_time;

namespace stage_common{
	/** Ajastinluokka, tarjoaa palvelun laskennassa kuluvan ajan mittaamiseen.
	Laskee mittauskutsujen välisten aikojen pituuksia ja soveltuu erityisesti
	toistuvien tapahtumien kuten pelimoottorin
	pelisilmukoiden pituuksien mittaamiseen.
	*/
	class Timer{
	public:
		/** Käynnistää ajastimen*/
		void start();
		/** Mittaa, kuinka paljon aikaa on kulunut edellisestä start()-
		kutsusta*/
		void stop();
		/** Mittaa, kuinka kauan on kulunut edellisestä start()-kutsusta ja
		käynnistää ajastimen uudestaan
		Käytännössa sama kuin stop(); start();*/
		void tick();
		/** Palauttaa tiedon siitä, kuinka paljon aikaa ajastin on
		olemassaolonsa aikana mitannut millisekunteina
		@returns	Ajastimen mittaama kokonaisaika (viimeisen stop() tai
				tick() -kutsun kohdalla)
		*/
		double totalTime(){ return totalms; }
		/** Palauttaa tiedon siitä, kuinka monta mittauspistettä ajastin on
		käsitellyt
		@returns	Kuinka monta stop() tai tick() -kutsua ajastin on
				käsitellyt
		*/
		unsigned int totalTicks(){ return ticks; }
		/** Palauttaa keskimäärin yhteen mittausväliin kuluneen ajan
		millisekunteina
		@returns	Ajastimen kokonaisaika jaettuna mittauspisteiden
				määrällä
		*/
		double averageTime(){ return (totalms / ticks); }
		/** Palauttaa edellisen mittausvälin pituuden millisekunteina
		@returns	Edellisen mittausvälin pituus
```

```
		*/
		double lastTickTime(){ return lastTick; }
	private:
		/** Ajastimen mittaama kokonaisaika*/
		double totalms = 0;
		/** Edellisen mittausvälin pituus*/
		double lastTick = 0;
		/** Mittausvälien kokonaismäärä*/
		unsigned int ticks = 0;
		/** Nykyisen mittausvälin käynnistysaika*/
		clock_time lastStart;
	};
}
#endif
```

Timer.cpp:

```
#include "stdafx.h"
#include "Timer.h"

using namespace stage_common;

typedef std::chrono::high_resolution_clock hrclock;

void Timer::start(){
	lastStart = hrclock::now();
}
void Timer::stop(){
	clock_time end = hrclock::now();
	std::chrono::duration<double, std::milli> duration = end - lastStart;
	totalms += duration.count();
	lastTick = duration.count();
	ticks++;
}
void Timer::tick(){
	stop();
	start();
}
```

Liite 2. Aktoripohjaisen Stage-pelimoottorin toteutus

Tämä liite sisältää aktoripohjaisen Stage-pelimoottoritoteutuksen ohjelmakoodin.

Moduuli Stage_core:

CameraComponent.h:

```
#ifndef CAMERACOMPONENT_H
#define CAMERACOMPONENT_H

/**Kamerakomponentin komponenttitunnus*/
#define CAMERA_ID 2

#include "stdafx.h"
#include <Component.h>
#include <Camera.h>
#include <Transform.h>
#include <CoreEvents.h>
#include <iostream>
#include "GraphicsControlActor.h"

namespace stage{
	/** Peliolioon liitettävä kamerakomponentti. Toimii wrapperina
stage_common::Camera:lle.
	Kun peliolioon liitetään tämä komponentti, pelimaailma voidaan piirtää sen
	kuvakulmasta.
	Ottaa vastaan viestit:
	Update (vastaa AllDone)
	Render (vastaa AllDone)
	CameraComponent::SetViewMatrix (vastaa AllDone)
	CamaraComponent::SetProjectionMatrix (vastaa AllDone)
	*/
	class CameraComponent : public Component{
		friend class GameLoop;
	public:
		//---Viestit---

		/** Viesti, joka pyytää kamerakomponenttia asettamaan uuden
		näkymämatriisin
		(Huom. normaalisti kamera hakee omistajaolioltaan näkymämatriisin
		joka Render-viestin yhteydessä)*/
		struct SetViewMatrix : public Event{
			/** Uusi näkymämatriisi*/
			glm::mat4& view;
			SetViewMatrix(uint64_t id, glm::mat4& view) : Event(id),
				view(view){}
		};
		/** Viesti, joka pyytää kamerakomponenttia asettamaan uuden
		projektiomatriisin */
		struct SetProjectionMatrix : public Event{
			/** Uusi projektiomatriisi*/
			glm::mat4& projection;
			SetProjectionMatrix(uint64_t id, glm::mat4& projection) :
				Event(id), projection(projection){}
		};
```

```
        //---Metodit---

        /** Luo uuden kamerakomponentin. Katso oikea käyttö yliluokasta.
        @param fw           Theron::Framework, jonka alaisuudessa tämä
        komponentti toimii
        @param owner  Sen peliolion osoite, joka omistaa tämän komponentin
        */
        CameraComponent(Theron::Framework& fw, Theron::Address owner);

        /** Hakee osoittimen tämän komponentin kameraolioon
        HUOM: ei säieturvallinen, älä käytä paluuarvoa pelimoottorin
        ollessa käynnissä
        @returns      Osoitin kameraolioon
        */
        stage_common::Camera* getRawCamera(){ return &cam; }
        /** Hakee olion komponenttitunnuksen
        @returns      Tämän komponentin tunnus
        */
        virtual int id(){ return CAMERA_ID; }
private:
        /** Onko komponentti käynnistetty, eli voiko se suorittaa update-
        ja render-kutsuja    */
        bool init = false;
        /** Komponentin kameraolio*/
        stage_common::Camera cam;
        /** Omistajaolion sijaintia ylläpitävän olion osoite*/
        Theron::Address transform;

        //---Metodit---

        /** Suorittaa loppuun komponentin käynnistyksen
        @param msg          Sijaintiolion komponenttitunnuksen sisältävä
                            viesti
        @param sender Sijaintiolion osoite
        */
        void initialize(const GameObject::ComponentFound &msg,
               Theron::Address sender);
        /** Suorittaa tarvittavan laskennan ruudun piirtoa varten
        @param msg          Renderöintipyyntö
        @param sender Lähettäjän osoite
        */
        virtual void render(const Render& msg, Theron::Address sender);
        /** Suorittaa loppuun tarvittavan laskennan ruudun piirtoa varten
        @param msg          Kameran uusi näkymämatriisi
        @param sender Lähettäjän osoite
        */
        void completeRender(const Transform::Matrix& msg, Theron::Address
               sender);
        /** Asettaa kameralle uuden näkymämatriisin
        @param msg          Kameran uusi näkymämatriisi
        @param sender Lähettäjän osoite
        */
        void setViewMatrix(const SetViewMatrix& msg, Theron::Address
               sender);
        /** Asettaa kameralle uuden projektiomatriisin
        @param msg          Kameran uusi projektiomatriisi
        @param sender Lähettäjän osoite
        */
        void setProjectionMatrix(const SetProjectionMatrix& msg,
               Theron::Address sender);
};
```

```
}
#endif
```

CameraComponent.cpp:

```
#include "stdafx.h"
#include "CameraComponent.h"
#include <LogActor.h>

using namespace stage;

//---Kontekstiyksikkö alkaa--
CameraComponent::CameraComponent(Theron::Framework& fw, Theron::Address owner) :
	Component(fw, owner){
	RegisterHandler(this, &CameraComponent::initialize);
	//Haetaan isäntäolion sijaintikomponentti, jotta kameran sijainti voidaan
	päivittää ruudun piirron yhteydessä
	uint64_t msgid = tracker.getNextID();
	EventContext& ev = tracker.addContext(0, msgid, Theron::Address::Null());
	ev.finalize = [](){};
	ev.error = [this](){
		LOGMSG("Error: Attempted to initialize camera component, but owner
			does not have a transform");
	};
	Send(GameObject::GetComponent(msgid, TRANSFORM_ID), owner);
	RegisterHandler(this, &CameraComponent::completeRender);
	RegisterHandler(this, &CameraComponent::setViewMatrix);
	RegisterHandler(this, &CameraComponent::setProjectionMatrix);

}

void CameraComponent::initialize(const GameObject::ComponentFound &msg,
	Theron::Address sender){
	if (!tracker.contains(msg.id)) return;
	tracker.decrement(msg.id);
	DeregisterHandler(this, &CameraComponent::initialize);
	transform = msg.component;
	init = true;
}
//---Kontekstiyksikkö päättyy--

//---Kontekstiyksikkö alkaa--
void CameraComponent::render(const Render& msg, Theron::Address sender){
	if (!init){
		//sijaintikomponenttia ei vielä saatu, joten kameran sijaintia ei
		voida päivittää
		Send(AllDone(msg.id), sender);
		return;
	}
	uint64_t id = tracker.getNextID();
	tracker.addContext(msg.id, id, sender);
	//Haetaan sijaintikomponentilta kameran sijainti tämän ruudunpäivityksen
	aikana
	Send(Transform::GetMatrix(id), transform);
}

void CameraComponent::completeRender(const Transform::Matrix& msg,
	Theron::Address sender){
	if (!tracker.contains(msg.id)) return;
	//Asetetaan kameralle uusi sijainti
	cam.setViewMatrix(msg.matrix);
```

```
	tracker.decrement(msg.id);
}
//---Kontekstiyksikkö päättyy--

void CameraComponent::setViewMatrix(const SetViewMatrix& msg, Theron::Address
	sender){
	cam.setViewMatrix(msg.view);
	Send(AllDone(msg.id), sender);
}

void CameraComponent::setProjectionMatrix(const SetProjectionMatrix& msg,
	Theron::Address sender){
	cam.setProjectionMatrix(msg.projection);
	Send(AllDone(msg.id), sender);
}
```

Gameloop.h:

```
#ifndef GAMELOOP_H
#define GAMELOOP_H

#include "stdafx.h"
#include <Theron\Framework.h>
#include <SceneManager.h>
#include "GraphicsControlActor.h"
#include <LogActor.h>
#include <EventChannelManager.h>

namespace stage {

	/** Olio, joka suorittaa pelimoottorin pelisilmukkaa ja hoitaa
	pelimoottorin yleiseen ylläpitoon kuuluvia asioita.*/
	class Gameloop : public SceneManager{
	public:
		/** Luo pelisilmukan ja avaa uuden OpenGL-ikkunan.
		@param windowName	Ikkunan nimi
		@param xres			Ikkunan vaakaresoluutio
		@param yres			Ikkunan pystyresoluutio
		*/
		Gameloop(std::string& windowName, int xres, int yres, uint32_t
			threadcount);
		/** Tuhoaa pelisilmukan	*/
		~Gameloop();
		/** Palauttaa pelisilmukan aikaskaalan (kuinka nopeasti
		simulaatiota suoritetaan)
		@returns	Aikaskaala liukulukuna
		*/
		float getTimescale();
		/** Asettaa pelisilmukan aikaskaalan (kuinka nopeasti simulaatiota
		suoritetaan)
		@param ts	Aikaskaala liukulukuna
		*/
		void setTimescale(float ts);
		/** Palauttaa viitteen pelimaailman olioita hallinnoivaan
		Theron::Framework-olioon*/
		Theron::Framework& getFramework(){ return fw; }
		/** Asettaa pelin pääkameran, jonka näkökulmasta pelimaailmaa
		kuvataan
		@param cam	Viite kameraolioon
		*/
		void setActiveCamera(stage_common::Camera* cam);
```

```
		/** Käynnistää pelisilmukan suorituksen, joka päättyy peliohjelman
		sulkeutuessa  */
		void start();
		/** Hakee viitteen tapahtumakanavia hallinnoivaan aktoriin
		@returns	Viite tapahtumakanavien hallinta-aktoriin
		*/
		EventChannelManager& getEventChannelManager(){ return
			eventChannelManager; }
	private:
		/** Pelin olioita ja niiden välisiä viestejä hallinnoiva olio*/
		Theron::Framework fw;
		/** Grafiikkamoottorin toiminnasta vastaava aktori*/
		GraphicsControlActor* gc;
		/** Lokista vastaava aktori*/
		LogActor* logger;
		/** Kamera, jonka kuvakulmasta pelimaailma piirretään*/
		stage_common::Camera* activeCam;
		/** Tapahtumakanavia huoltava aktori*/
		EventChannelManager eventChannelManager;
		/** Pelisilmukan aikaskaala (kuinka nopeasti simulaatiota
		suoritetaan)*/
		float timescale = 1;
		/** Pysäytetäänkö pelisilmukan suoritus tämän suorituskerran
		jälkeen*/
		bool abort = false;
		/** Seuraavan lähetettävän viestin tunnus*/
		uint32_t msgid = 0;

		/** Pelisilmukkametodi: kutsuu toistuvasti peliolioiden update- ja
		render-metodeja.*/
		void loop();
		/** Pysäyttää pelisilmukan suorituksen*/
		void stop();
		/** Viimeistelee pelisilmukan pysäyttämisen ja jättää pelisilmukan
		tilaan, jossa se voidaan turvallisesti tuhota.*/
		void shutdown();
	};
}
#endif
```

Gameloop.cpp:

```
#pragma once

#include "stdafx.h"
#include <Theron\Theron.h>
#include <iostream>
#include <string>
#include "Gameloop.h"
#include "GameObject.h"
#include <CoreEvents.h>
#include "CameraComponent.h"
#include <Timer.h>
#include <Input.h>

using namespace stage;

Gameloop::Gameloop(std::string& windowName, int xres, int yres, uint32_t
	threadcount): fw(Theron::Framework::Parameters(threadcount)),
	eventChannelManager(fw){
	if (SceneManager::globalManager != Theron::Address::Null()){
```

```
                //Sallitaan vain yksi Gameloop
                std::abort();
        }
        gc = new GraphicsControlActor(fw, windowName, xres, yres);
        logger = new LogActor(fw);
        SceneManager::globalManager = receiver.GetAddress();
}
Gameloop::~Gameloop(){
        if (globalManager == receiver.GetAddress()) globalManager =
                Theron::Address::Null();
        //Poistetaan pelialueet
        for (std::vector<Scene*>::iterator i = scenes.begin(); i != scenes.end();
                i++){
                delete *i;
        }
        //Poistetaan grafiikkamoottori
        delete gc;
        //Poistetaan lokipalvelu
        delete logger;
}
float Gameloop::getTimescale() {
                return timescale;
}
void Gameloop::setTimescale(float ts) {
        timescale = ts;
}
void Gameloop::setActiveCamera(stage_common::Camera* cam){
        activeCam = cam;
}
void Gameloop::start() {
        if (Theron::Address::Null() == activeScene){
                fw.Send(LogActor::LogError("Failed to start game engine: active
                        scene not set"), receiver.GetAddress(), logger
                        ->GetAddress());
                return;
        }
        if (activeCam == nullptr){
                fw.Send(LogActor::LogError("Failed to start game engine: active
                        camera not set"), receiver.GetAddress(), logger
                        ->GetAddress());
                return;
        }
        loop();
}
void Gameloop::stop(){
        abort = true;
}
void Gameloop::loop() {
        Theron::Address sender;

        //Placeholder-viestit catchereitä varten
        AllDone adMSG(0);
        SetActiveScene sasMSG(0,0);
        CreateScene csMSG(0);

        Theron::Address recAddress = receiver.GetAddress();
        stage_common::Timer upTimer;       //Päivitysajastin
        stage_common::Timer rendTimer;     //Piirtoajastin
        stage_common::Timer maintTimer;    //Ylläpitoajastin
        stage_common::Timer loopTimer;     //Koko pelisilmukan ajastin
```

```
//Pelisilmukka
while (!abort) {
	loopTimer.start();

	//Päivitysvaihe
	upTimer.start();
	uint64_t id = Event::generateID(recAddress, msgid++);
	fw.Send(Update((float)loopTimer.lastTickTime() * timescale, id),
		recAddress, activeScene);
	while (doneCatcher.Empty()){
		//Odotetaan, kunnes saadaan viesti
		receiver.Wait();

		//Luodaan uudet pelialueet
		while (!createSceneCatcher.Empty()){
			createSceneCatcher.Pop(csMSG, sender);
			Theron::Address newScene = createScene();
			fw.Send(NewScene(csMSG.id, scenes.size() - 1,
				newScene), recAddress, sender);
		}

		//Vaihdetaan pelialuetta
		while (!setSceneCatcher.Empty()){
			setSceneCatcher.Pop(sasMSG, sender);
			if (setActiveScene(sasMSG.scene)){
				fw.Send(AllDone(sasMSG.id), recAddress,
					sender);
			}
			else fw.Send(Error(sasMSG.id), recAddress, sender);
		}
	}
	doneCatcher.Pop(adMSG, sender);
	upTimer.stop();

	//Piirtovaihe
	rendTimer.start();
	//Haetaan piirrettävät mallit
	id = Event::generateID(recAddress, msgid++);
	fw.Send(Render(id), recAddress, activeScene);
	while (doneCatcher.Empty()){
		receiver.Wait();
	}
	doneCatcher.Pop(adMSG, sender);
	//Piirretään kuva ruudulle (tehtävä pääsäikeessä, koska OpenGL-
	//kontekstit ovat säiekohtaisia)
	gc->getRawController()->draw(*activeCam);
	rendTimer.stop();

	//Ylläpitovaihe
	maintTimer.start();
	stage_common::Input::getSingleton().update(false);

	//Huolletaan tapahtumakanavat
	id = Event::generateID(recAddress, msgid++);
	fw.Send(EventChannelManager::ChannelMaintenance(id), recAddress,
		eventChannelManager.GetAddress());
	while (doneCatcher.Empty()){
		receiver.Wait();
	}
	doneCatcher.Pop(adMSG, sender);
```

```
			//Tarkistetaan, pitääkö ohjelma sulkea
			if (!abortCatcher.Empty()) stop();
			if (gc->shouldClose()) stop();
			maintTimer.stop();
			loopTimer.stop();
		}
		//Kirjoitetaan lokiin suorituskykytiedot
		fw.Send(LogActor::LogMessage("Total runtime " +
			std::to_string(loopTimer.totalTime())), recAddress, logge
			r->GetAddress());
		fw.Send(LogActor::LogMessage("Total frames: " +
			std::to_string(loopTimer.totalTicks())), recAddress, logger
			->GetAddress());
		fw.Send(LogActor::LogMessage("Average loop time: " +
			std::to_string(loopTimer.averageTime())), recAddress, logger
			->GetAddress());
		fw.Send(LogActor::LogMessage("Average fps: " + std::to_string(1000 /
			loopTimer.averageTime())), recAddress, logger->GetAddress());
		fw.Send(LogActor::LogMessage("Average update time: " +
			std::to_string(upTimer.averageTime())), recAddress, logger
			->GetAddress());
		fw.Send(LogActor::LogMessage("Average render time: " +
				std::to_string(rendTimer.averageTime())), recAddress,
				logger->GetAddress());
		fw.Send(LogActor::LogMessage("Average maintenance time: " +
				std::to_string(maintTimer.averageTime())), recAddress,
				logger->GetAddress());
		shutdown();
}
void Gameloop::shutdown(){
		//Annetaan lokiolion kirjoittaa kaikki vielä odottavat viestit, jotta se
		//voi sulkeutua turvallisesti
		fw.Send(LogActor::LogMessage("Shutting down"), receiver.GetAddress(),
			logger->GetAddress());
		Theron::Catcher<LogActor::Terminate> terminateCatcher;
		receiver.RegisterHandler(&terminateCatcher,
			&Theron::Catcher<LogActor::Terminate>::Push);
		fw.Send(LogActor::Terminate(), receiver.GetAddress(), logger-
			>GetAddress());
		while (terminateCatcher.Empty()){
			//Odotetaan, kunnes lokioliolla ei enää ole tehtävää
			receiver.Wait();
		}
}
```

GraphicsControlActor.h:

```
#ifndef GRAPHICSCONTROLACTOR_H
#define GRAPHICSCONTROLACTOR_H

#include "stdafx.h"
#include <GraphicsController.h>
#include <CoreEvents.h>
#include <Model.h>
#include <Camera.h>
#include <glm\glm.hpp>
#include <iostream>

namespace stage {
		/** Grafiikkamoottoria hallinnoiva aktori-singleton.
		Pitää huolen, että lista piirrettävistä 3D-malleista muodostetaan
```

```
säieturvallisesti.
Ottaa vastaan viestit:
GraphicsControlActor::Queue (vastaa AllDone)
*/
class GraphicsControlActor : public Theron::Actor{
	friend class Gameloop;
public:
	/** Viesti, joka lisää uuden 3D-mallin seuraavassa
	ruudunpäivityksessä piirrettävien mallien listalle*/
	struct Queue : public Event{
		/** Piirrettävä 3D-malli*/
		stage_common::Model* model;
		/** Pelimaailman piste, johon malli piirretään  */
		const glm::mat4& position;
		Queue(Theron::Address originator, uint32_t msgID,
			stage_common::Model* mod, glm::mat4& pos):
			Event(originator, msgID), model(mod), position(pos){}
		Queue(uint64_t msgID, stage_common::Model* mod, const
			glm::mat4& pos) :
			Event(msgID), model(mod), position(pos){}
	};
	/** Luo uuden grafiikkamoottoria hallinnoivan aktorin ja avaa peli-
	ikkunan
	@param fw			Tätä aktoria hallinnoiva
					Theron::Framework
	@param windowname		Luotavan peli-ikkunan nimi
	@param x			Ikkunan vaakaresoluutio
	@param y			Ikkunan pystyresoluutio
	*/
	GraphicsControlActor(Theron::Framework& fw, std::string windowname,
		int x, int y);
	/** Tuhoaa grafiikkamoottoria hallinnoivan aktorin*/
	~GraphicsControlActor();
	/** Antaa globaalin GraphicsControlActor-singletonin osoitteen
	@returns	Globaalin grafiikkamoottoria hallinnoivan aktorin
			osoite
	*/
	static Theron::Address getGlobalController(){ return
		globalController; }
private:
	/** Globaalin grafiikkamoottoria hallinnoivan aktorin osoite
	*/
	static Theron::Address globalController;
	/** Grafiikkamoottoriolio
	*/
	stage_common::GraphicsController gc;

	/** Antaa osoittimen varsinaiseen grafiikkamoottoriolioon.
	Tarvitaan kuvan piirtämistä varten, koska OpenGL-kontekstit ovat
	säiekohtaisia,
	joten 3D-mallit on piirrettävä siinä säikeessä, joka alun perin
	käynnisti grafiikkamoottorin.
	@returns	Osoitin grafiikkamoottoriolioon
	*/
	stage_common::GraphicsController* getRawController() { return
		&gc; }
	/** Kertoo, onko käyttäjä pyytänyt ohjelmaa pysähtymään.
	@returns	True, jos ohjelman suoritus tulisi pysäyttää
	*/
	bool shouldClose(){ return gc.shouldStop(); }
	/** Lisää uuden 3D-mallin seuraavassa ruudunpäivityksessä
```

```
		piirrettävien mallien listalle
		@param msg	Viesti, joka sisältää osoittimen piirrettävään
				malliin sekä sijainnin, johon malli piirretään
		*/
		void queue(const Queue& msg, Theron::Address sender);
	};
}
#endif
```

GraphicsControlActor.cpp:

```
#include "stdafx.h"
#include "GraphicsControlActor.h"

using namespace stage;

//Alkuarvo globaalin singletonin osoitteelle
Theron::Address GraphicsControlActor::globalController = Theron::Address::Null();

GraphicsControlActor::GraphicsControlActor(Theron::Framework& fw, std::string
	windowname, int x, int y) : Theron::Actor(fw), gc(windowname, x, y) {
	if (globalController == Theron::Address::Null()){
		globalController = this->GetAddress();
	}
	else {
		//Sallitaan vain yksi grafiikkamoottoriolio
		std::cout << "Global GraphicsControlActor already set\n";
		abort();
	}
	RegisterHandler(this, &GraphicsControlActor::queue);
}
GraphicsControlActor::~GraphicsControlActor(){
	if (globalController == this->GetAddress()) globalController =
		Theron::Address::Null();
}
void GraphicsControlActor::queue(const Queue& msg, Theron::Address sender){
	gc.queue(msg.model, msg.position);
	Send(AllDone(msg.id), sender);
}
```

ModelComponent.h:

```
#ifndef MODELCOMPONENT_H
#define MODELCOMPONENT_H

/**3D-mallikomponentin komponenttitunnus*/
#define MODEL_ID 3

#include "stdafx.h"
#include <Model.h>
#include <Component.h>
#include <Transform.h>
#include "GraphicsControlActor.h"
#include <iostream>

namespace stage{
	/** Peliolioon liitettävä 3D-mallikomponentti.
	Kun olioon liitetään tämä komponentti, sen sijaintiin piirretään 3D-malli
	joka ruudunpäivityksellä.
	Jokaisella piirrettävällä pelioliolla tulee olla oma mallikomponentti,
	mutta usea mallinkomponentti
	voi käyttää samaa 3D-mallioliota.
```

```
	Ottaa vastaan viestit:
	Update (vastaa AllDone)
	Render (vastaa AllDone)
	*/
	class ModelComponent : public Component{
	public:
		/** Luo uuden mallikomponentin. Katso oikea käyttö yliluokasta.
		@param fw		Theron::Framework, jonka alaisuudessa tämä
					komponentti toimii
		@param mod		Osoitin siihen 3D-malliin, joka halutaan
					piirtää
		@param owner		Sen peliolion osoite, joka omistaa tämän
					komponentin
		*/
		ModelComponent(Theron::Framework& fw, stage_common::Model* mod,
			Theron::Address owner);
		/** Hakee olion komponenttitunnuksen
		@returns	Tämän komponentin tunnus
		*/
		virtual int id(){ return MODEL_ID; }
	private:
		/** Onko komponentti käynnistetty, eli voiko se suorittaa update-
		ja render-kutsuja	*/
		bool init = false;
		/** Osoitin siihen 3D-malliin, joka piirretään, kun tälle oliolle
		lähetetään Render-viesti*/
		stage_common::Model* mod;
		/** Omistajaolion sijaintia ylläpitävän olion osoite*/
		Theron::Address transform;

		/** Suorittaa loppuun komponentin käynnistyksen
		@param msg		Sijaintiolion komponenttitunnuksen sisältävä
					viesti
		@param sender Sijaintiolion osoite
		*/
		void initialize(const GameObject::ComponentFound& msg,
			Theron::Address sender);
		/** Piirtää ruudulle tämän komponentin 3D-mallin
		@param msg		Renderöintipyyntö
		@param sender Lähettäjän osoite
		*/
		void render(const Render& msg, Theron::Address sender);
		/** Suorittaa loppuun tarvittavan laskennan mallin piirtoa varten
		@param msg		3D-mallin sijainti pelimaailmassa
		@param sender Lähettäjän osoite
		*/
		void completeRender(const Transform::Matrix& msg, Theron::Address
			sender);
	};
}
#endif
```

ModelComponent.cpp:

```
#include "stdafx.h"
#include "ModelComponent.h"
#include <LogActor.h>

using namespace stage;

//---Kontekstiyksikkö alkaa--
```

```
ModelComponent::ModelComponent(Theron::Framework& fw, stage_common::Model* mod,
	Theron::Address owner) : Component(fw, owner), mod(mod){
	RegisterHandler(this, &ModelComponent::initialize);
	//Haetaan isäntäolion sijaintikomponentti, jotta malli voidaan piirtää
	oikeaan paikkaan
	uint64_t msgid = tracker.getNextID();
	EventContext& ev = tracker.addContext(0, msgid, Theron::Address::Null());
	ev.finalize = [](){};
	ev.error = [this](){
		LOGMSG("Error: Attempted to initialize model component, but owner
			does not have a transform");
	};
	Send(GameObject::GetComponent(msgid, TRANSFORM_ID), owner);
	RegisterHandler(this, &ModelComponent::completeRender);
}
void ModelComponent::initialize(const GameObject::ComponentFound& msg,
	Theron::Address sender){
	if (!tracker.contains(msg.id)) return;
	tracker.decrement(msg.id);
	transform = msg.component;
	DeregisterHandler(this, &ModelComponent::initialize);
	init = true;
}
//---Kontekstiyksikkö päättyy--

//---Kontekstiyksikkö alkaa--
void ModelComponent::render(const Render& msg, Theron::Address sender){
	if (!init) return;
	uint64_t id = tracker.getNextID();
	tracker.addContext(msg.id, id, sender);
	//Haetaan isäntäolion nykyinen sijainti, jotta malli voidaan piirtää
	//oikeaan paikkaan
	Send(Transform::GetMatrix(id), transform);
}
void ModelComponent::completeRender(const Transform::Matrix& msg, Theron::Address
	sender){
	if (!tracker.contains(msg.id)) return;
	Send(GraphicsControlActor::Queue(msg.id, mod, msg.matrix),
		GraphicsControlActor::getGlobalController());
}
//---Kontekstiyksikkö päättyy--
```

PhysicsComponent.h:

```
#ifndef PHYSICSCOMPONENT_H
#define PHYSICSCOMPONENT_H

#define PHYSICSCOMPONENT_ID 4

#include "stdafx.h"
#include <Component.h>
#include <Collisions.h>
#include <Collider.h>
#include <SphereCollider.h>
#include <AABBCollider.h>
#include <Transform.h>
#include <EventChannel.h>
#include <LogActor.h>
#include <unordered_set>

namespace stage{
```

```
/** Komponentti, joka tekee isäntäoliostaan fysiikkaolion, joka liikkuu
pelimaailmassa vakionopeudella
ja voi törmätä muihin fysiikkaolioihin sekä staattiseen tasogeometriaan
Käsittelee viestit:
Update (palauttaa AllDone)
Render (palauttaa AllDone)
PhysicsComponent::CollisionCheck (palauttaa AllDone)
PhysicsComponent::CollisionDetected (palauttaa AllDone jos törmäys on jo
käsittelyssä tai FinishCollision)
PhysicsComponent::StaticCollision (palauttaa AllDone)
*/
class PhysicsComponent : public Component{
public:
	//---Viestit---

	/** Viesti, joka ilmoittaa lähettäneen fysiikkaolion sijainnin
	muuttuneen ja pyytää muita fysiikkaolioita
	tarkistamaan mahdolliset törmäykset	*/
	struct CollisionCheck : public Event{
		/** Lähettäjän törmäyshahmo*/
		const stage_common::Collider& coll;
		/** Alkuperäisen lähettäjän osoite*/
		const Theron::Address originator;
		/** Edellisestä ruudunpäivityksestä kulunut aika
		millisekunteina*/
		float elapsedMS;
		CollisionCheck(uint64_t id, stage_common::Collider& coll,
			Theron::Address originator, float elapsedMS)
			: Event(id), coll(coll), originator(originator),
			elapsedMS(elapsedMS){}
	};
	/** Viesti, joka ilmoittaa, että vastaanottaja on törmännyt
	lähettäjään*/
	struct CollisionDetected : public Event{
		/** Lähettäjän törmäyshahmo */
		const stage_common::Collider& coll;
		/** Lähettäjän nopeus*/
		glm::vec3 otherVelocity;
		/** Lähettäjän massa*/
		float otherMass;
		CollisionDetected(uint64_t id, stage_common::Collider& coll,
			glm::vec3 otherVelocity, float otherMass)
			: Event(id), coll(coll),
			otherVelocity(otherVelocity), otherMass(otherMass){}
	};
	/** Viesti, joka ilmoittaa, että vastaanottaja on törmännyt
	liikkumattomaan peliolioon*/
	struct StaticCollision : public Event{
		/** Lähettäjän törmäyshahmo*/
		const stage_common::Collider& coll;
		StaticCollision(uint64_t id, stage_common::Collider& coll) :
			Event(id), coll(coll){}
	};
	/** Viesti, joka ilmoittaa törmäyksen käsittelyn onnistuneen*/
	struct FinishCollision : public Event{
		/** Vastaanottajan nopeuteen tehtävä muutos*/
		glm::vec3 velocityAdjustment;
		FinishCollision(uint64_t id, glm::vec3 velAdj) : Event(id),
			velocityAdjustment(velAdj){}
	};
```

```
        //---Metodit---

        /** Luo uuden fysiikkakomponentin pallotörmäyshahmolla
        @param fw            Komponenttia hallinnoiva Theron::Framework
        @param owner         Komponentin omistavan peliolion osoite
        @param transform     Isäntäolion sijaintia ylläpitävän komponentin
                             osoite
        @param radius        Pallotörmäyshahmon säde
        @param initialV      Fysiikkaolion liikesuunta ja nopeus
                             simulaation alussa
        @param mass          Fysiikkaolion massa
        @param collisionEventChannel      Fysiikkaolioiden
                             törmäystarkistusviestien käyttämä
                             tapahtumakanava
        */
        PhysicsComponent(Theron::Framework& fw, Theron::Address owner,
               Theron::Address transform,
               float radius, glm::vec3 initialV, float mass,
               EventChannel<CollisionCheck>& collisionEventChannel);
        /** Luo uuden fysiikkakomponentin AABB-törmäyshahmolla
        @param fw            Komponenttia hallinnoiva Theron::Framework
        @param owner         Komponentin omistavan peliolion osoite
        @param transform     Isäntäolion sijaintia ylläpitävän komponentin
                             osoite
        @param size          Törmäyshahmon koko
        @param initialV      Fysiikkaolion liikesuunta ja nopeus
                             simulaation alussa
        @param mass          Fysiikkaolion massa
        @param collisionEventChannel      Fysiikkaolioiden
                             törmäystarkistusviestien käyttämä
                             tapahtumakanava
        */
        PhysicsComponent(Theron::Framework& fw, Theron::Address owner,
               Theron::Address transform,
               glm::vec3 size, glm::vec3 initialV, float mass,
               EventChannel<CollisionCheck>& collisionEventChannel);
        /** Tuhoaa fysiikkakomponentin*/
        ~PhysicsComponent();
        /** Hakee fysiikkakomponentin komponenttitunnuksen
        @returns      Komponentin tunnus
        */
        virtual int id(){ return PHYSICSCOMPONENT_ID; }
private:
        //---Kentät---
        /** Viite fysiikkamoottorin viestien käyttämään viestikanavaan*/
        EventChannel<CollisionCheck>& collisionEventChannel;
        /** Fysiikkaolion törmäyshahmo*/
        stage_common::Collider* collider;
        /** Kopio fysiikkaolion törmäyshahmosta.
        Käytetään törmäystentunnistuksessa varmistamaan, että törmäyshahmon
        muokkaaminen ei aiheuta rinnakkaisuusongelmia*/
        stage_common::Collider* tempCollider;
        /** Isäntäolion sijaintia ylläpitävän olion osoite*/
        Theron::Address transform;
        /** Fysiikkaolion liikesuunta ja nopeus*/
        glm::vec3 velocity;
        /** Fysiikkaolion sijainti ruudunpäivityksen alussa*/
        glm::vec3 oldPos;
        /** Fysiikkaolion massa*/
        float mass;
        /** Onko fysiikkaolion tila päivitetty nykyisen ruudunpäivityksen
```

```
aikana*/
bool updatedThisFrame = false;
/** Onko komponentin alustus suoritettu loppuun*/
bool init = false;
/** Niiden fysiikkaolioiden joukko, joihin on törmätty tämän
ruudunpäivityksen aikana.
Käytetään estämään saman törmäyksen käsitteleminen kahdesti, jos
kumpikin fysiikkaolio
käsittelee sitä samanaikaisesti rinnakkain.*/
std::unordered_set<int> collidedThisFrame;

//---Metodit---

/** Molemmille konstruktoreille yhteiset alustukset suorittava
metodi
@returns      Konteksti-ID, jota käytetään alustusviestien
              tunnistamiseen
*/
uint64_t setup();
/** Suoritetaan komponentin alustus loppuun ja asetetaan
törmäyshahmoksi pallo
@param msg           Fysiikkaolion sijainnin sisältävä viesti
@param sender Viestin lähettäjä
*/
void finishSphereSetup(const Transform::Position& msg,
       Theron::Address sender);
/** Suoritetaan komponentin alustus loppuun ja asetetaan
törmäyshahmoksi AABB (Axis-Aligned Bounding Box)
@param msg           Fysiikkaolion sijainnin sisältävä viesti
@param sender Viestin lähettäjä
*/
void finishAABBSetup(const Transform::Position& msg,
       Theron::Address sender);

//--Kontekstiyksikkö alkaa--

/** Päivitetään tila uutta ruudunpäivitystä vastaavaksi
@param up     Päivityspyyntö
@param from   Pyynnön lähettäjä
*/
void update(const Update &up, Theron::Address from);
/** Suorittaa loppuun tilanpäivityksen päivittämällä muuttuneen
sijainnin sijaintikomponenttiin
@param msg    Isäntäolion nykyisen sijainnin ilmoittava viesti
@param from   Viestin lähettäjä
*/
void finishUpdate(const Transform::Position& msg, Theron::Address
       from);

//--Kontekstiyksikkö päättyy--

//--Kontekstiyksikkö alkaa--

/**Tarkistaa, onko tämä fysiikkolio törmännyt toiseen
@param msg    Toisen fysiikkaolion törmäyshahmon sisältävä viesti
@param from   Viestin lähettäjä
*/
void collisionCheck(const CollisionCheck& msg, Theron::Address
       from);
/** Suorittaa loppuun törmäystapahtuman käsittelyn
@param msg    Viesti, joka ilmoittaa törmäystapahtuman käsittelyn
```

```
                        päättyneen ja sisältää törmäyksen aiheuttaman
                        muutoksen tämän kappaleen nopeuteen
                @param from   Viestin lähettäjä
                */
                void finishCollision(const FinishCollision& msg, Theron::Address
                        from);

                //--Kontekstiyksikkö päättyy--

                /** Käsittelee havaitun törmäyksen kahden fysiikkaolion välillä
                @param msg    Törmäyksestä kertova viesti
                @param from   Toinen törmännyt fysiikkaolio
                */
                void collisionDetected(const CollisionDetected& msg,
                        Theron::Address from);
                /** Käsittelee havaitun törmäyksen fysiikkaolion ja staattisen
                törmäyshahmon välillä
                @param msg    Törmäyksestä kertova viesti
                @param from   Staattinen törmäyshahmokomponentti, johon törmättiin
                */
                void staticCollision(const StaticCollision& msg, Theron::Address
                        from);
                /** Siirtää törmäyshahmoa fysiikkaolion nykyisen nopeuden
                mukaisesti
                @param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
                                     millisekunteina
                */
                void updatePosition(float elapsedMS);
                /** Suorittaa pelisilmukan piirtovaiheen laskennan
                @param rend   Piirtopyyntöviesti
                @param from   Viestin lähettäjä
                */
                void render(const Render &rend, Theron::Address from);
        };
}
#endif
```

PhysicsComponent.cpp:

```
#include "stdafx.h"
#include "PhysicsComponent.h"

using namespace stage;

PhysicsComponent::PhysicsComponent(Theron::Framework& fw, Theron::Address owner,
        Theron::Address transform,
        float radius, glm::vec3 initialV, float mass,
        EventChannel<CollisionCheck>& collisionEventChannel) :
        Component(fw, owner), transform(transform), velocity(initialV),
        mass(mass), collisionEventChannel(collisionEventChannel){
        //Rekisteröidään viestinkäsittelijä olion alustuksen lopettavalle
        //metodille
        RegisterHandler(this, &PhysicsComponent::finishSphereSetup);
        //Konstruktoreille yhteiset alustukset
        uint64_t id = setup();
        //Tallennetaan törmäyshahmon säde kontekstimuuttujaan 0
        tracker.setVariable<float>(id, 0, radius);
        //Pyydetään sijaintioliolta nykyinen sijainti
        Send(Transform::GetPosition(id), transform);
        //Suoritus jatkuu metodissa finishSphereSetup
}
```

```
PhysicsComponent::PhysicsComponent(Theron::Framework& fw, Theron::Address owner,
	Theron::Address transform,
	glm::vec3 size, glm::vec3 initialV, float mass,
	EventChannel<CollisionCheck>& collisionEventChannel) :
	Component(fw, owner), transform(transform), velocity(initialV),
	mass(mass), collisionEventChannel(collisionEventChannel){
	//Rekisteröidään viestinkäsittelijä olion alustuksen lopettavalle
	//metodille
	RegisterHandler(this, &PhysicsComponent::finishAABBSetup);
	//Konstruktoreille yhteiset alustukset
	uint64_t id = setup();
	//Tallennetaan törmäyshahmon koko kontekstimuuttujaan 0
	tracker.setVariable<glm::vec3>(id, 0, size);
	//Pyydetään sijaintioliolta nykyinen sijainti
	Send(Transform::GetPosition(id), transform);
	//Suoritus jatkuu metodissa finishAABBSetup
}
PhysicsComponent::~PhysicsComponent(){
	delete collider;
	delete tempCollider;
}

uint64_t PhysicsComponent::setup(){
	//Rekisteröidään viestinkäsittelijät
	RegisterHandler(this, &PhysicsComponent::collisionCheck);
	RegisterHandler(this, &PhysicsComponent::collisionDetected);
	RegisterHandler(this, &PhysicsComponent::staticCollision);
	RegisterHandler(this, &PhysicsComponent::finishCollision);
	//Luodaan uusi viestikonteksti alustuksen loppuunsuorittamista varten
	uint64_t id = tracker.getNextID();
	EventContext& context = tracker.addContext(0, id,
		Theron::Address::Null());
	context.finalize = [](){};
	context.error = context.finalize;
	//Rekisteröidään fysiikkaolio tapahtumakanavan viestien vastaanottajaksi
	Send(EventChannel<CollisionCheck>::RegisterRecipient(this->GetAddress()),
		collisionEventChannel.GetAddress());
	return id;
}
void PhysicsComponent::finishSphereSetup(const Transform::Position& msg,
	Theron::Address sender){
	DeregisterHandler(this, &PhysicsComponent::finishSphereSetup);
	RegisterHandler(this, &PhysicsComponent::finishUpdate);
	//Luodaan törmäyshahmo kontekstimuuttujasta haettavalla säteellä ja
	//viestin sisältämällä sijainnilla
	collider = new
		stage_common::SphereCollider(tracker.getVariable<float>(msg.id, 0),
		msg.position);
	oldPos = msg.position;
	init = true;
	tracker.decrement(msg.id);
}
void PhysicsComponent::finishAABBSetup(const Transform::Position& msg,
	Theron::Address sender){
	DeregisterHandler(this, &PhysicsComponent::finishAABBSetup);
	RegisterHandler(this, &PhysicsComponent::finishUpdate);
	//Luodaan törmäyshahmo kontekstimuuttujasta haettavalla koolla ja viestin
	//sisältämällä sijainnilla
	collider = new
	stage_common::AABBCollider(tracker.getVariable<glm::vec3>(msg.id, 0),
		msg.position);
```

```
	oldPos = msg.position;
	init = true;
	tracker.decrement(msg.id);
}

//--Kontekstiyksikkö alkaa--
void PhysicsComponent::update(const Update &up, Theron::Address from){
	if (!init){
		//Ei tehdä mitään, ellei tilaa ole vielä alustettu loppuun
		Send(AllDone(up.id), from);
		return;
	}
	//Päivitetään sijainti vain jos sitä ei vielä ole tehty törmäyskäsittelyn
	//seurauksena
	if (!updatedThisFrame) updatePosition(up.elapsedMS);
	//Luodaan konteksti törmäystarkistusviesteille
	uint64_t id = tracker.getNextID();
	EventContext& context = tracker.addContext(up.id, id, from, 0);
	context.finalize = [this, &context](){
		//Kun kaikki törmäyksen on tarkistettu, luodaan uusi konteksti
		//sijainnin päivittämiselle
		uint64_t id = tracker.getNextID();
		tracker.addContext(context.getOriginalID(), id,
			context.getOriginalSender());
		//Haetaan isäntäolion sijainti uudestaan, koska muut komponentit
		//ovat ehkä tehneet siihen muutoksia
		Send(Transform::GetPosition(id), transform);
		//Suoritus jatkuu metodissa finishUpdate
	};
	//Lähetetään törmäystarkistusviestit kaikille tapahtumakanavaan
	//rekisteröityneille
	CollisionCheck newmsg(id, *tempCollider, this->GetAddress(),
		up.elapsedMS);
	const std::list<Theron::Address>& recipients =
		collisionEventChannel.getRecipients();
	bool sent = false;
	for (std::list<Theron::Address>::const_iterator i = recipients.cbegin(); i
		!= recipients.cend(); i++){
		if ((*i) != this->GetAddress()){
			tracker.trackedSend<CollisionCheck>(up.id, newmsg, *i,
				from);
			sent = true;
		}
	}
	//Jos ei lähetettäviä viestejä, suljetaan tarpeeton konteksti
	if (!sent){
		context.finalize();
		tracker.remove(id);
		//Suoritus jatkuu metodissa finishUpdate
	}
}
void PhysicsComponent::finishUpdate(const Transform::Position& msg,
	Theron::Address from){
	//Lasketaan nykyisen ruudunpäivityksen aikana liikuttu matka
	glm::vec3 translation = collider->center - oldPos;
	//Päivitetään törmäyshahmon sijainti
	collider->center = oldPos = msg.position + translation;
	//Päivitetään sijaintikomponentti Translate-viestillä, jotta muiden
	//komponenttien mahdollisesti tekemiä
	//muutoksia ei peruta
	Send(Transform::Translate(msg.id, translation), transform);
```

```
	//Sijaintikomponentti vastaa AllDone-viestillä, joka sulkee nykyisen
	//kontekstin ja
	//lähettää automaattisesti AllDone-viestin isäntäoliolle
}
//--Kontekstiyksikkö päättyy--

//--Kontekstiyksikkö alkaa--
void PhysicsComponent::collisionCheck(const CollisionCheck& msg, Theron::Address
	from){
	if (!init){
		//Ei tehdä mitään, jos komponenttia ei ole alustettu loppuun
		Send(AllDone(msg.id), from);
		return;
	}
	//Päivitetään sijainti, jos sitä ei vielä ole päivitetty Update-viestin
	//seurauksena
	if (!updatedThisFrame) updatePosition(msg.elapsedMS);
	//Ei tehdä mitään, jos toinen olio käsittelee jo samaa törmäystä
	//rinnakkain tai oliot eivät ole törmänneet
	if (collidedThisFrame.count(msg.originator.AsInteger()) || !tempCollider
		->checkCollision(msg.coll)) Send(AllDone(msg.id), from);
	else {
		//Merkitään, että tämä törmäys on jo käsittelyssä, jos toisen olion
		//osoite on omaa suurempi
		//Tällä varmistetaan, että törmäys käsitellään tasan kerran
		if (this->GetAddress().AsInteger() < msg.originator.AsInteger())
			collidedThisFrame.insert(msg.originator.AsInteger());
		//Luodaan uusi konteksti törmäyksen käsittelyä varten
		uint64_t id = tracker.getNextID();
		tracker.addContext(msg.id, id, from);
		Send(CollisionDetected(id, *tempCollider, velocity, mass),
			msg.originator);
		//Suoritus jatkuu metodissa FinishCollision
	}
}
void PhysicsComponent::finishCollision(const FinishCollision& msg,
	Theron::Address from){
	if (!tracker.contains(msg.id)){
		LOGERR("Warning: Unknown collision finish event received from " +
			std::to_string(from.AsInteger()));
		return;
	}
	velocity += msg.velocityAdjustment;
	//Vastausviesti törmäystapahtuman aiheuttajalle lähetetään automaattisesti
	//kontekstin sulkeutuessa
	tracker.decrement(msg.id);
}
//--Kontekstiyksikkö päättyy--

void PhysicsComponent::collisionDetected(const CollisionDetected& msg,
	Theron::Address from){
	//Ei tehdä mitään, jos toinen olio käsittelee jo samaa törmäystä
	//rinnakkain
	if (collidedThisFrame.count(from.AsInteger())) Send(AllDone(msg.id),
		from);
	else {
		//Merkitään, että tämä törmäys on jo käsittelyssä, jos toisen olion
		//osoite on omaa suurempi
		//Tällä varmistetaan, että törmäys käsitellään tasan kerran
		if (this->GetAddress().AsInteger() < from.AsInteger())
			collidedThisFrame.insert(from.AsInteger());
```

```
                glm::vec3 otherNewV = msg.otherVelocity;
                //Lasketaan törmäyksestä aiheutuvat nopeuden muutokset
                stage_common::Collisions::collisionVelocityChange(velocity, mass,
                        otherNewV, msg.otherMass);
                //Siirretään tämän olion törmäyshahmoa, kunnes oliot eivät enää
                //törmää
                stage_common::Collisions::backOff(*collider, -1.0f * velocity,
                        msg.coll);
                //Tiedotetaan toiselle oliolle sen nopeuteen tulleesta muutoksesta
                Send(FinishCollision(msg.id, otherNewV - msg.otherVelocity), from);
        }
}
void PhysicsComponent::staticCollision(const StaticCollision& msg,
        Theron::Address from){
        //Lasketaan törmäyksestä aiheutuvat nopeuden muutokset
        velocity = stage_common::Collisions::reflect(velocity,
                msg.coll.getCollisionNormal(*tempCollider, velocity));
        //Siirretään tämän olion törmäyshahmoa, kunnes oliot eivät enää törmää
        stage_common::Collisions::backOff(*collider, -1.0f * velocity, msg.coll);
        Send(AllDone(msg.id), from);
}
void PhysicsComponent::updatePosition(float elapsedMS){
        collider->center = oldPos + (velocity * elapsedMS);
        updatedThisFrame = true;
        delete tempCollider;
        //Luodaan virtuaalikopiometodilla törmäyshahmosta väliaikainen kopio,
        //jonka tilaa voidaan turvallisesti lukea
        //muista säikeistä
        tempCollider = collider->copy();
}
void PhysicsComponent::render(const Render &rend, Theron::Address from){
        updatedThisFrame = false;
        collidedThisFrame.clear();
        Send(AllDone(rend.id), from);
}
```

StaticGeometryComponent.h:

```
#pragma once

#ifndef STATICGEOMETRYCOMPONENT_H
#define STATICGEOMETRYCOMPONENT_H

#define STATICGEOMETRYCOMPONENT_ID 5

#include "PhysicsComponent.h"

namespace stage{
        /** Luokka, joka esittää staattista pelimaailman tasogeometriaa, kuten
        maastoa tai seiniä.
        Käsittelee viestit:
        Update (palauttaa AllDone)
        Render (palauttaa AllDone)
        PhysicsComponent::CollisionCheck (palauttaa AllDone)
        */
        class StaticGeometryComponent : public Component{
        public:
                /** Luo uuden staattisen törmäyskomponentin pallotörmäyshahmolla
                @param fw               Komponenttia hallinnoiva Theron::Framework
                @param owner            Komponentin omistavan peliolion osoite
                @param transform        Isäntäolion sijaintia ylläpitävän komponentin
```

```
                        osoite
        @param radius       Pallotörmäyshahmon säde
        @param collisionEventChannel      Fysiikkaolioiden
                        törmäystarkistusviestien käyttämä
                        tapahtumakanava
        */
        StaticGeometryComponent(Theron::Framework& fw, Theron::Address
                owner, float radius, Theron::Address transform,
                Theron::Address collisionEventChannel);
        /** Luo uuden staattisen törmäyskomponentin AABB-törmäyshahmolla
        (Axis-aligned bounding box)
        @param fw           Komponenttia hallinnoiva Theron::Framework
        @param owner        Komponentin omistavan peliolion osoite
        @param transform    Isäntäolion sijaintia ylläpitävän komponentin
                        osoite
        @param radius       AABB-törmäyshahmon koko
        @param collisionEventChannel      Fysiikkaolioiden
                        törmäystarkistusviestien käyttämä
                        tapahtumakanava
        */
        StaticGeometryComponent(Theron::Framework& fw, Theron::Address
                owner, glm::vec3 size, Theron::Address transform,
                Theron::Address collisionEventChannel);
        /** Hakee staatisen törmäyshahmokomponentin komponenttitunnuksen
        @returns      Komponentin tunnusluku
        */
        int id(){ return STATICGEOMETRYCOMPONENT_ID; }
        /** Tuhoaa staattisen törmäyshahmokomponentin*/
        ~StaticGeometryComponent();
private:
        /** Komponentin törmäyshahmo*/
        stage_common::Collider* collider;
        /** Komponentin isäntäolion sijaintia ylläpitävän olion Theron-
        osoite*/
        Theron::Address transform;
        /** Törmäysviestikanavan osoite*/
        Theron::Address collisionEventChannel;
        /** Onko tämän komponentin alustus suoritettu loppuun*/
        bool init = false;

        //--Kontekstiyksikkö alkaa--

        /** Päivittää komponentin tilan
        @param up           Tilanpäivityspyyntö
        @param sende  Pyynnön lähettäjä
        */
        void update(const Update &up, Theron::Address from);
        /** Suorittaa tilanpäivityksen loppuun
        @param msg    Peliolion sijainnin sisältävä viesti
        @param from   Viestin lähettäjä
        */
        void finishUpdate(const Transform::Position& msg, Theron::Address
                from);

        //--Kontekstiyksikkö päättyy--

        /** Tarkistaa onko fysiikkaolio törmännyt tähän olioon
        @param msg    Viesti, joka sisältää fysiikkaolion törmäyshahmon
                        tiedot
        @param from   Viestin lähettäjä
        */
```

```
		void collisionCheck(const PhysicsComponent::CollisionCheck& msg,
			Theron::Address from);
		/** Suoritetaan komponentin alustus loppuun ja asetetaan
		törmäyshahmoksi pallo
		@param msg		Peliolion sijainnin sisältävä viesti
		@param sender Viestin lähettäjä
		*/
		void finishSphereSetup(const Transform::Position& msg,
		Theron::Address sender);
		/** Suoritetaan komponentin alustus loppuun ja asetetaan
		törmäyshahmoksi AABB (Axis-Aligned Bounding Box)
		@param msg		Peliolion sijainnin sisältävä viesti
		@param sender Viestin lähettäjä
		*/
		void finishAABBSetup(const Transform::Position& msg,
		Theron::Address sender);
		/** Molemmille konstruktoreille yhteiset alustukset suorittava
		metodi
		@returns	Konteksti-ID, jota käytetään alustusviestien
				tunnistamiseen
		*/
		uint64_t setup();
	};
}
#endif
```

StaticGeometryComponent.cpp

```
#include "stdafx.h"
#include "StaticGeometryComponent.h"

using namespace stage;

StaticGeometryComponent::StaticGeometryComponent(Theron::Framework& fw,
	Theron::Address owner, float radius, Theron::Address transform,
	Theron::Address collisionEventChannel) : Component(fw, owner),
	transform(transform), collisionEventChannel(collisionEventChannel){
	//Rekisteröidään käsittelijä alustuksen suorittamiselle loppuun
	RegisterHandler(this, &StaticGeometryComponent::finishSphereSetup);
	//Konstruktoreille yhteiset alustukset
	uint64_t id = setup();
	//Talletetaan törmäyshahmon säde kontekstimuuttujaan
	tracker.setVariable<float>(id, 0, radius);
	Send(Transform::GetPosition(id), transform);
	//Suoritus jatkuu metodissa finishSphereSetup
}
StaticGeometryComponent::StaticGeometryComponent(Theron::Framework& fw,
	Theron::Address owner, glm::vec3 size, Theron::Address transform,
	Theron::Address collisionEventChannel) : Component(fw, owner),
	transform(transform), collisionEventChannel(collisionEventChannel){
	//Rekisteröidään käsittelijä alustuksen suorittamiselle loppuun
	RegisterHandler(this, &StaticGeometryComponent::finishAABBSetup);
	//Konstruktoreille yhteiset alustukset
	uint64_t id = setup();
	//Talletetaan törmäyshahmon koko kontekstimuuttujaan
	tracker.setVariable<glm::vec3>(id, 0, size);
	Send(Transform::GetPosition(id), transform);
	//Suoritus jatkuu metodissa finishAABBSetup
}
StaticGeometryComponent::~StaticGeometryComponent(){
	delete collider;
```

```
}

//--Kontekstiyksikkö alkaa--
void StaticGeometryComponent::update(const Update &up, Theron::Address from){
	if (!init){
		//Ei tehdä mitään, jos alustusta ei vielä ole suoritettu loppuun
		Send(AllDone(up.id), from);
		return;
	}
	//Haetaan uudestaan peliolion sijainti, sillä jokin muu komponentti on
	//ehkä muuttanut sitä
	uint64_t id = tracker.getNextID();
	tracker.addContext(up.id, id, from);
	Send(Transform::GetPosition(id), transform);
	//Suoritus jatkuu metodissa finishUpdate
}
void StaticGeometryComponent::finishUpdate(const Transform::Position& msg,
	Theron::Address from){
	//Päivitetään törmäyshahmon sijainti
	collider->center = msg.position;
	//Poistetaan konteksti, jolloin vastausviesti Update-viestiin lähtee
	automaattisesti
	tracker.decrement(msg.id);
}
//--Kontekstiyksikkö päättyy--

void StaticGeometryComponent::collisionCheck(const
	PhysicsComponent::CollisionCheck& msg, Theron::Address from){
	if (!init){
		//Ei tehdä mitään, jos alustusta ei vielä ole suoritettu loppuun
		Send(AllDone(msg.id), from);
		return;
	}
	//Ei tehdä mitään, jos törmäystä ei ole tapahtunut
	if (!collider->checkCollision(msg.coll)) Send(AllDone(msg.id), from);
	else {
		//Luodaan konteksti törmäyksen käsittelyä varten
		uint64_t id = tracker.getNextID();
		tracker.addContext(msg.id, id, from);
		//Lähetetään ilmoitus törmäyksestä
		Send(PhysicsComponent::StaticCollision(id, *collider),
			msg.originator);
	}
}
void StaticGeometryComponent::finishSphereSetup(const Transform::Position& msg,
	Theron::Address sender){
	DeregisterHandler(this, &StaticGeometryComponent::finishSphereSetup);
	RegisterHandler(this, &StaticGeometryComponent::finishUpdate);
	//Luodaan törmäyshahmo kontekstimuuttujasta haettavalla säteellä ja
	//viestin sisältämällä sijainnilla
	collider = new
		stage_common::SphereCollider(tracker.getVariable<float>(msg.id, 0),
		msg.position);
	init = true;
	tracker.decrement(msg.id);
}
void StaticGeometryComponent::finishAABBSetup(const Transform::Position& msg,
	Theron::Address sender){
	DeregisterHandler(this, &StaticGeometryComponent::finishAABBSetup);
	RegisterHandler(this, &StaticGeometryComponent::finishUpdate);
	//Luodaan törmäyshahmo kontekstimuuttujasta haettavalla koolla ja viestin
```

```
		//sisältämällä sijainnilla
		collider = new
			stage_common::AABBCollider(tracker.getVariable<glm::vec3>(msg.id,
			0), msg.position);
		init = true;
		tracker.decrement(msg.id);
}
uint64_t StaticGeometryComponent::setup(){
		RegisterHandler(this, &StaticGeometryComponent::collisionCheck);
		//Luodaan uusi viestikonteksti alustuksen loppuunsuorittamista varten
		uint64_t id = tracker.getNextID();
		EventContext& context = tracker.addContext(0, id,
			Theron::Address::Null());
		context.finalize = [](){};
		context.error = context.finalize;
		//Rekisteröidään peliolio törmäystapahtumakanavan viestien
		//vastaanottajaksi
		Send(EventChannel<PhysicsComponent::CollisionCheck>::RegisterRecipient(
			this->GetAddress()), collisionEventChannel);
		return id;
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"
#include <Theron\Theron.h>

#define WIN32_LEAN_AND_MEAN
```

Moduuli Stage_demo:

CameraControlComponent.h:

```
#include "stdafx.h"

#ifndef CAMERACONTROLCOMPONENT_H
#define CAMERACONTROLCOMPONENT_H

#include <Component.h>
#include <Transform.h>
#include <Input.h>
#include <GLFW\glfw3.h>
#include <SceneManager.h>

#define CAMERACONTROLCOMPONENT_ID 6
#define CAMERASPEED 0.025f

namespace stage{
		/** Kameran hallintaan tarkoitettu komponentti, joka mahdollistaa
		isäntäolionsa liikuttamisen näppäimistöllä
		Käsittelee viestit:
		Update (palauttaa AllDone)
		Render (palauttaa AllDone)
		*/
		class CameraControlComponent : public Component{
		public:
			/** Luo uuden kameranhallintakomponentin
			@param fw		Komponenttia hallinnoiva Theron::Framework
			@param owner		Komponentin isäntäolion Theron-osoite
```

```
        @param transform     Isäntäolion sijaintia ylläpitävän komponentin
                             Theron-osoite
        */
        CameraControlComponent(Theron::Framework& fw, Theron::Address
                owner, Theron::Address transform)
                : Component(fw, owner), transform(transform){
                stage_common::Input& in =
                        stage_common::Input::getSingleton();
                //Rekisteröidään ne näppäimet, joiden tila halutaan lukea
                //Voidaan tehdä säieturvallisesti, koska Input-olion
                //näppäinlistaan
                //haetaan arvot vasta ylläpitovaiheessa
                in.registerKey(GLFW_KEY_W);
                in.registerKey(GLFW_KEY_S);
                in.registerKey(GLFW_KEY_A);
                in.registerKey(GLFW_KEY_D);
                in.registerKey(GLFW_KEY_R);
                in.registerKey(GLFW_KEY_F);
                in.registerKey(GLFW_KEY_ESCAPE);
        }
        /** Hakee kameranhallintakomponentin komponenttitunnuksen
        @returns      Komponentin tyyppitunnus
        */
        virtual int id(){ return CAMERACONTROLCOMPONENT_ID; }
private:
        /** Isäntäolion sijaintia ylläpitävän komponentin Theron-osoite
        */
        Theron::Address transform;
        /** Päivittää komponentin tilan
        @param msg           Päivityspyyntö
        @param sender Pyynnön lähettäjä
        */
        void update(const Update& msg, Theron::Address sender){
                stage_common::Input& in =
                        stage_common::Input::getSingleton();
                //Isäntäolion liikettä kuvaava vektori
                glm::vec3 movement;
                //muutetaan liikevektoria pohjassa olevien näppäinten
                //perusteella
                //Voidaan tehdä säieturvallisesti, koska Input-olion
                //näppäinlistan
                //arvoja muutetaan vain ylläpitovaiheessa
                if (in.getKeyDown(GLFW_KEY_W)) movement.z += CAMERASPEED *
                        msg.elapsedMS;
                if (in.getKeyDown(GLFW_KEY_S)) movement.z -= CAMERASPEED  *
                        msg.elapsedMS;
                if (in.getKeyDown(GLFW_KEY_A)) movement.x += CAMERASPEED  *
                        msg.elapsedMS;
                if (in.getKeyDown(GLFW_KEY_D)) movement.x -= CAMERASPEED  *
                        msg.elapsedMS;
                if (in.getKeyDown(GLFW_KEY_F)) movement.y += CAMERASPEED  *
                        msg.elapsedMS;
                if (in.getKeyDown(GLFW_KEY_R)) movement.y -= CAMERASPEED  *
                        msg.elapsedMS;
                //Lopetetaan suoritus, jos Esc pohjassa
                if (in.getKeyDown(GLFW_KEY_ESCAPE))
                        Send(SceneManager::Abort(),
                                SceneManager::getGlobalManager());
                //Luodaan konteksti isäntäolion siirtämistä varten
                uint64_t id = tracker.getNextID();
                tracker.addContext(msg.id, id, sender);
```

```
                        //Pyydetään isäntäoliota siirtymään
                        Send(Transform::Translate(id, movement), transform);
                        //Vastausviesti lähetetään automaattisesti, kun Transform
                        //vastaa
                }
        };
}
#endif
```

GameObjectFactory.h:

```
#ifndef GAMEOBJECTFACTORY_H
#define GAMEOBJECTFACTORY_H

#include "stdafx.h"
#include "Plane.h"
#include "Sphere.h"
#include "Waiter.h"
#include <Scene.h>
#include <glm\glm.hpp>
#include <PhysicsComponent.h>
#include <StaticGeometryComponent.h>
#include <ModelComponent.h>
#include <SimpleShader.h>

namespace stage{
        /** Tehdasluokka, joka valmistaa pelimoottorin demo-ohjelman käyttämiä
        peliolioita*/
        class GameObjectFactory{
        public:
                /** Rakentaa satunnaiseen paikkaan pallon, joka lähtee liikkumaan
                satunnaiseen suuntaan
                @param fw                       Pelimoottorin käyttämä Theron::Framework
                @param scene                    Pelialue, johon pallo luodaan
                @param maxCoordinates           Pallon maksimietäisyys pelimaailman
                                                origosta
                @param collisionEventChannel        Fysiikkaolioiden käyttämä
                                                tapahtumakanava
                @returns                        Luodun peliolion Theron-osoite
                */
                Theron::Address constructRandomSphere(Theron::Framework& fw,
                        Theron::Address scene, glm::vec3 maxCoordinates,
                        EventChannel<PhysicsComponent::CollisionCheck>&
                        collisionEventChannel, int waitLimit){
                        //Placeholder-muuttujat vastaanotettaville viesteille
                        Scene::NewObject obj(0, Theron::Address::Null());
                        Theron::Address temp;
                        //Pyydetään pelialuetta luomaan uusi peliolio
                        fw.Send(Scene::CreateObject(0), rec.GetAddress(), scene);
                        //Odotetaan olion valmistumista
                        rec.Wait();
                        catcher.Pop(obj, temp);
                        //Arvotaan pallolle sijainti
                        glm::vec3 translation(randomFloat(-maxCoordinates.x,
                                maxCoordinates.x),
                                randomFloat(-maxCoordinates.y, maxCoordinates.y),
                                randomFloat(-maxCoordinates.z, maxCoordinates.z));
                        glm::mat4 transform = glm::translate(glm::mat4(1.0f),
                                translation);
                        //Luodaan pallolle sijaintikomponentti
                        Transform* tf = new Transform(fw, obj.object, transform);
```

```
			//Liitetään palloon 3D-malli
			ModelComponent* mod = new ModelComponent(fw,
				&(getSingleton().mod_sphere), obj.object);
			//Arvotaan pallolle nopeus
			glm::vec3 velocity(randomFloat(-0.01f, 0.01f), randomFloat(-
				0.01f, 0.01f), randomFloat(-0.01f, 0.01f));
			//Luodaan pallolle fysiikkakomponentti
			PhysicsComponent* pc = new PhysicsComponent(fw, obj.object,
				tf->GetAddress(), 1.0f, velocity, 1.0f,
				collisionEventChannel);
			if (waitLimit > 0) Waiter* w = new Waiter(fw, obj.object,
				waitLimit);
			return obj.object;
		}
		/** Rakentaa haluttuun pelimaailman sijaintiin seinän
		@param fw			Pelimaailmaa hallinnoiva Theron::Framework
		@param scene		Sen pelialueen osoite, johon seinä rakennetaan
		@param transform	Seinän sijaintia kuvaava 4x4-matriisi
		@param size			Seinän koko
		@param collisionEventChannel		Fysiikkaolioiden käyttämä
											tapahtumakanava
		@returns			Luodun peliolion Theron-osoite
		*/
		Theron::Address constructWall(Theron::Framework& fw,
			Theron::Address scene, glm::mat4& transform, glm::vec3 size,
			Theron::Address collisionEventChannel){
			//Placeholder-muuttujat vastaanotettaville viesteille
			Scene::NewObject obj(0, Theron::Address::Null());
			Theron::Address temp;
			//Pyydetään pelialuetta luomaan uusi peliolio
			fw.Send(Scene::CreateObject(0), rec.GetAddress(), scene);
			//Odotetaan olion valmistumista
			rec.Wait();
			catcher.Pop(obj, temp);
			//Luodaan seinälle sijaintikomponentti
			Transform* tf = new Transform(fw, obj.object, transform);
			//Liitetään seinään 3D-malli
			ModelComponent* mod = new ModelComponent(fw,
				&(getSingleton().mod_plane), obj.object);
			//Liitetään seinään törmäyshahmo
			StaticGeometryComponent* sgc = new
				StaticGeometryComponent(fw, obj.object, size, tf
				->GetAddress(), collisionEventChannel);
			return obj.object;
		}
		/** Hakee viitteen globaaliin GameObjectFactory-instanssiin
		@returns	Viite GameObjectFactory-singletoniin
		*/
		static GameObjectFactory& getSingleton(){
			static GameObjectFactory gof;
			return gof;
		}
	private:
		/** Demon 3D-mallien käyttämä sävytinohjelma*/
		stage_common::SimpleShader ss;
		/** Pallon 3D-malli*/
		stage_common::Model mod_sphere;
		/** Seinän 3D-malli*/
		stage_common::Model mod_plane;
		/** Olio, jota tehdasolio käyttää vastaanottamaan viestejä
		aktoreilta*/
```

```
		Theron::Receiver rec;
		/** Olio, jota tehdasolio käyttää käsittelemään aktoreilta
		vastaanotetut viestit*/
		Theron::Catcher<Scene::NewObject> catcher;
		/** Apufunktio, joka arpoo satunnaisen liukuluvun kahden luvun
		väliltä
		@param start  Arvottavan luvun alaraja
		@param end		Arvottavan luvun yläraja
		@returns		Staunnainen luku ala- ja ylärajan väliltä
		*/
		static float randomFloat(float start, float end){
			float random = ((float)rand()) / (float)RAND_MAX;
			return start + (end - start) * random;
		}
		/** Luo pelioliotehtaan*/
		GameObjectFactory() : mod_sphere(generate_sphere_vertices(),
			generate_sphere_colors(), &ss),
			mod_plane(generate_plane_vertices(),
			generate_plane_colors(), &ss){
			rec.RegisterHandler(&catcher,
			&Theron::Catcher<Scene::NewObject>::Push);
		}
	};
}
#endif
```

Plane.h:

```
#ifndef PLANE_H
#define PLANE_H

#include "stdafx.h"
#include <vector>
#include <glm\glm.hpp>

//Sisältää apufunktioita, joiden avulla voidaan muodostaa litteän tason
//piirtämiseen vaadittava 3D-malli
namespace stage{
	/** Litteän tason verteksit*/
	static std::vector<glm::vec3> plane_vertices = {
		glm::vec3(-1.000000, 0.000000, 1.000000),
		glm::vec3(1.000000, 0.000000, 1.000000),
		glm::vec3(-1.000000, 0.000000, -1.000000),
		glm::vec3(1.000000, 0.000000, -1.000000)
	};
	/** Litteän tason tahkot*/
	static std::vector<int> plane_faces = {
		2, 4, 3,
		1, 2, 3
	};
	/** Yhdistää tason verteksit ja tahkot yhtenäiseksi verteksilistaksi
	@returns	Lista litteän tason vektoreista
	*/
	static std::vector<glm::vec3> generate_plane_vertices(){
		std::vector<glm::vec3> ret;
		for (unsigned int i = 0; i < plane_faces.size(); i++){
			ret.push_back(plane_vertices[plane_faces[i] - 1]);
		}
		return ret;
	}
	/** Arpoo litteän tason vertekseille värit
```

```
	@returns	Lista värivektoreita
	*/
	static std::vector<glm::vec3> generate_plane_colors(){
		std::vector<glm::vec3> ret;
		for (unsigned int i = 0; i < plane_faces.size(); i++){
			ret.push_back(glm::vec3(static_cast <float> (rand()) /
			static_cast <float> (RAND_MAX),
			static_cast <float> (rand()) / static_cast <float>
			(RAND_MAX),
			static_cast <float> (rand()) / static_cast <float>
			(RAND_MAX)));
		}
		return ret;
	}
}
#endif
```

Sphere.h:

```
#ifndef SPHERE_H
#define SPHERE_H

#include "stdafx.h"
#include <vector>
#include <glm\glm.hpp>

//Sisältää apufunktioita, joiden avulla voidaan muodostaa pallon piirtämiseen
//vaadittava 3D-malli
namespace stage{
	/** Pallon verteksit*/
	static std::vector<glm::vec3> sphere_vertices = {
		glm::vec3(0.000000, -1.000000, 0.000000),
		glm::vec3(0.723607, -0.447220, 0.525725),
		glm::vec3(-0.276388, -0.447220, 0.850649),
		glm::vec3(-0.894426, -0.447216, 0.000000),
		glm::vec3(-0.276388, -0.447220, -0.850649),
		glm::vec3(0.723607, -0.447220, -0.525725),
		glm::vec3(0.276388, 0.447220, 0.850649),
		glm::vec3(-0.723607, 0.447220, 0.525725),
		glm::vec3(-0.723607, 0.447220, -0.525725),
		glm::vec3(0.276388, 0.447220, -0.850649),
		glm::vec3(0.894426, 0.447216, 0.000000),
		glm::vec3(0.000000, 1.000000, 0.000000),
		glm::vec3(-0.162456, -0.850654, 0.499995),
		glm::vec3(0.425323, -0.850654, 0.309011),
		glm::vec3(0.262869, -0.525738, 0.809012),
		glm::vec3(0.850648, -0.525736, 0.000000),
		glm::vec3(0.425323, -0.850654, -0.309011),
		glm::vec3(-0.525730, -0.850652, 0.000000),
		glm::vec3(-0.688189, -0.525736, 0.499997),
		glm::vec3(-0.162456, -0.850654, -0.499995),
		glm::vec3(-0.688189, -0.525736, -0.499997),
		glm::vec3(0.262869, -0.525738, -0.809012),
		glm::vec3(0.951058, 0.000000, 0.309013),
		glm::vec3(0.951058, 0.000000, -0.309013),
		glm::vec3(0.000000, 0.000000, 1.000000),
		glm::vec3(0.587786, 0.000000, 0.809017),
		glm::vec3(-0.951058, 0.000000, 0.309013),
		glm::vec3(-0.587786, 0.000000, 0.809017),
		glm::vec3(-0.587786, 0.000000, -0.809017),
		glm::vec3(-0.951058, 0.000000, -0.309013),
```

```
        glm::vec3(0.587786, 0.000000, -0.809017),
        glm::vec3(0.000000, 0.000000, -1.000000),
        glm::vec3(0.688189, 0.525736, 0.499997),
        glm::vec3(-0.262869, 0.525738, 0.809012),
        glm::vec3(-0.850648, 0.525736, 0.000000),
        glm::vec3(-0.262869, 0.525738, -0.809012),
        glm::vec3(0.688189, 0.525736, -0.499997),
        glm::vec3(0.162456, 0.850654, 0.499995),
        glm::vec3(0.525730, 0.850652, 0.000000),
        glm::vec3(-0.425323, 0.850654, 0.309011),
        glm::vec3(-0.425323, 0.850654, -0.309011),
        glm::vec3(0.162456, 0.850654, -0.499995)
};
/** Pallon tahkot*/
static std::vector<int> sphere_faces = {
        1, 14, 13,
        2, 14, 16,
        1, 13, 18,
        1, 18, 20,
        1, 20, 17,
        2, 16, 23,
        3, 15, 25,
        4, 19, 27,
        5, 21, 29,
        6, 22, 31,
        2, 23, 26,
        3, 25, 28,
        4, 27, 30,
        5, 29, 32,
        6, 31, 24,
        7, 33, 38,
        8, 34, 40,
        9, 35, 41,
        10, 36, 42,
        11, 37, 39,
        39, 42, 12,
        39, 37, 42,
        37, 10, 42,
        42, 41, 12,
        42, 36, 41,
        36, 9, 41,
        41, 40, 12,
        41, 35, 40,
        35, 8, 40,
        40, 38, 12,
        40, 34, 38,
        34, 7, 38,
        38, 39, 12,
        38, 33, 39,
        33, 11, 39,
        24, 37, 11,
        24, 31, 37,
        31, 10, 37,
        32, 36, 10,
        32, 29, 36,
        29, 9, 36,
        30, 35, 9,
        30, 27, 35,
        27, 8, 35,
        28, 34, 8,
        28, 25, 34,
```

```
			25, 7, 34,
			26, 33, 7,
			26, 23, 33,
			23, 11, 33,
			31, 32, 10,
			31, 22, 32,
			22, 5, 32,
			29, 30, 9,
			29, 21, 30,
			21, 4, 30,
			27, 28, 8,
			27, 19, 28,
			19, 3, 28,
			25, 26, 7,
			25, 15, 26,
			15, 2, 26,
			23, 24, 11,
			23, 16, 24,
			16, 6, 24,
			17, 22, 6,
			17, 20, 22,
			20, 5, 22,
			20, 21, 5,
			20, 18, 21,
			18, 4, 21,
			18, 19, 4,
			18, 13, 19,
			13, 3, 19,
			16, 17, 6,
			16, 14, 17,
			14, 1, 17,
			13, 15, 3,
			13, 14, 15,
			14, 2, 15
		};
		/** Yhdistää pallon verteksit ja tahkot yhtenäiseksi verteksilistaksi
		@returns	Lista pallon vektoreista
		*/
		static std::vector<glm::vec3> generate_sphere_vertices(){
			std::vector<glm::vec3> ret;
			for (unsigned int i = 0; i < sphere_faces.size(); i++){
				ret.push_back(sphere_vertices[sphere_faces[i] - 1]);
			}
			return ret;
		}
		/** Arpoo pallon vertekseille värit
		@returns	Lista värivektoreita
		*/
		static std::vector<glm::vec3> generate_sphere_colors(){
			std::vector<glm::vec3> ret;
			for (unsigned int i = 0; i < sphere_faces.size(); i++){
				ret.push_back(glm::vec3(static_cast <float> (rand()) /
				static_cast <float> (RAND_MAX),
				static_cast <float> (rand()) / static_cast <float>
				(RAND_MAX),
				static_cast <float> (rand()) / static_cast <float>
				(RAND_MAX)));
			}
			return ret;
		}
}
```

```
#endif
```

simplefragmentshader.glsl:

```
//Pikselisävytin
#version 330 core

//Parametri: värit
in vec3 fragmentColor;
//Palautetaan lasketut värit
out vec3 color;

void main(){
    color = fragmentColor;
}
```

simplevertexshader.glsl:

```
//Verteksisävytin
#version 330 core

//1. parametri: verteksit
layout(location = 0) in vec3 vertexPosition_modelspace;
//2. parametri: värit
layout(location = 1) in vec3 vertexColor;

//Annetaan värit pikselisävyttimelle
out vec3 fragmentColor;

//Mallin sijainti suhteessa kameraan
uniform mat4 MVP;

//Lasketaan verteksin sijainti ruudulla suhteessa malliin ja annetaan värit
//pikselisävyttimelle
void main(){
    vec4 v = vec4(vertexPosition_modelspace,1);
    gl_Position = MVP * v;
       fragmentColor = vertexColor;
}
```

Stage_demo.cpp:

```
#include "stdafx.h"
#include <Theron\Theron.h>
#include <string>
#include <iostream>
#include <Gameloop.h>
#include <GameObject.h>
#include <Component.h>
#include <CoreEvents.h>
#include <Scene.h>
#include <functional>
#include <Transform.h>
#include <SimpleShader.h>
#include <ModelComponent.h>
#include <CameraComponent.h>
#include <EventChannel.h>
#include <PhysicsComponent.h>
#include <StaticGeometryComponent.h>
#include "Plane.h"
#include "Sphere.h"
#include "CameraControlComponent.h"
#include "GameObjectFactory.h"
```

```
#include <fstream>

using namespace stage;

/** Stage-moottorin demo-ohjelman pääohjelmametodi
Luo suljetun kuution, jonka sisälle generoidaan satunnaisesti törmäileviä palloja
*/
int _tmain(int argc, _TCHAR* argv[])
{
	char c;
	//Kuution koko
	int SCALE = 10;
	//Pallojen määrä
	int SPHERES = 5;
	//Säikeiden määrä
	uint32_t THREADS = 16;
	int WAIT = 0;
	std::string configfile;
	std::ifstream configStream("config.ini", std::ios::in);
	//Luetaan parametrit konfiguraatiotiedostosta
	if (configStream.is_open())
	{
		std::string line = "";
		std::string start, end;
		int delimiterPos;
		while (getline(configStream, line)){
			delimiterPos = line.find("=");
			//Rivillä ei "="-merkkiä
			if (delimiterPos == std::string::npos){
				std::cerr << "Invalid configuration parameter " <<
					start << std::endl;
				continue;
			}
			start = line.substr(0, delimiterPos);
			end = line.substr(delimiterPos + 1);
			//SCALE-parametri
			if (start == "SCALE"){
				try{
					SCALE = std::stoi(end);
					if (SCALE < 5) SCALE = 5;
				}
				catch (...){
					std::cerr << "Error parsing configuration
						parameter SCALE" << std::endl;
					continue;
				}
			}
			//SPHERES-parametri
			else if (start == "SPHERES"){
				try{
					SPHERES = std::stoi(end);
					if (SPHERES < 1) SPHERES = 1;
				}
				catch (...){
					std::cerr << "Error parsing configuration
						parameter SPHERES" << std::endl;
					continue;
				}
			}
			//THREADS-parametri
			else if (start == "THREADS"){
```

```
                        try{
                                THREADS = std::stoi(end);
                                if (THREADS < 1) THREADS = 1;
                        }
                        catch (...){
                                std::cerr << "Error parsing configuration
                                        parameter THREADS" << std::endl;
                                continue;
                        }
                }
                //WAIT-parametri
                else if (start == "WAIT"){
                        try{
                                WAIT = std::stoi(end);
                                if (WAIT < 0) WAIT = 0;
                        }
                        catch (...){
                                std::cerr << "Error parsing configuration
                                        parameter WAIT" << std::endl;
                                continue;
                        }
                }
                //Muut parametrit
                else std::cerr << "Unknown configuration parameter " <<
                        start << std::endl;
        }
        configStream.close();
}
else std::cerr << "Warning: config.ini not found, falling back to default
        parameters" << std::endl;

//Luodaan pelisilmukka
stage::Gameloop loop(std::string("Stage engine demo"), 640, 480, THREADS);
Theron::Framework& fw = loop.getFramework();
//Luodaan tapahtumakanava fysiikkaolioille
EventChannel<PhysicsComponent::CollisionCheck> collChannel(fw);
loop.getEventChannelManager().addChannel(collChannel.GetAddress());
//Luodaan pelialue
Theron::Address sc = loop.createScene();
//Asetetaan luotu pelialue aktiiviseksi
loop.setActiveScene(0);

//Oliot, joiden avulla vastaanotetaan pelimaailman alustuksessa syntyvät
//viestit
Theron::Receiver rec;
Theron::Receiver adRec;
Theron::Catcher<Scene::NewObject> catcher;
Theron::Catcher<AllDone> adCatcher;
rec.RegisterHandler(&catcher, &Theron::Catcher<Scene::NewObject>::Push);
adRec.RegisterHandler(&adCatcher, &Theron::Catcher<AllDone>::Push);


Scene::NewObject camobject(0, Theron::Address::Null());
Theron::Address temp;
//Luodaan kameraolio
fw.Send(Scene::CreateObject(0), rec.GetAddress(), sc);
rec.Wait();
catcher.Pop(camobject, temp);
//Kameran sijaintiolio
Transform* tr1 = new Transform(fw, camobject.object);
fw.Send(Transform::SetMatrix(0, glm::translate(glm::mat4(1.0f),
```

```
        glm::vec3(0, 0, -SCALE * 2))), adRec.GetAddress(), tr1
        ->GetAddress());

glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, SCALE *
        10.0f);
glm::mat4 View = glm::lookAt(
        glm::vec3(0, 0, -SCALE * 2),
        glm::vec3(0, 0, 0),
        glm::vec3(0, 1, 0)
);
//Kameran kamerakomponentti
CameraComponent* cam = new CameraComponent(fw, camobject.object);
fw.Send(CameraComponent::SetViewMatrix(0, View), adRec.GetAddress(), cam
        ->GetAddress());
fw.Send(CameraComponent::SetProjectionMatrix(0, Projection),
        adRec.GetAddress(), cam->GetAddress());
//Asetetaan luotu kamera aktiiviseksi
loop.setActiveCamera(cam->getRawCamera());
//Kameran näppäimistöohjaus
CameraControlComponent* camcc = new CameraControlComponent(fw,
        camobject.object, tr1->GetAddress());

GameObjectFactory& factory = GameObjectFactory::getSingleton();

//Kuution seinät

//Kuution pohja
glm::mat4 bottompos;
bottompos = glm::scale(bottompos, glm::vec3(SCALE, 1, SCALE));
bottompos = glm::translate(bottompos, glm::vec3(0, -SCALE, 0));
factory.constructWall(fw, sc, bottompos, glm::vec3(SCALE, 0, SCALE),
        collChannel.GetAddress());
//Kuution katto
glm::mat4 toppos;
toppos = glm::rotate(toppos, glm::radians(180.0f), glm::vec3(1.0f, 0.0f,
        0.0f));
toppos = glm::translate(toppos, glm::vec3(0, -SCALE, 0));
toppos = glm::scale(toppos, glm::vec3(SCALE, 1, SCALE));
factory.constructWall(fw, sc, toppos, glm::vec3(SCALE, 0, SCALE),
        collChannel.GetAddress());
//Kuution vasen seinä
glm::mat4 leftpos;
leftpos = glm::rotate(leftpos, glm::radians(90.0f), glm::vec3(0.0f, 0.0f,
        1.0f));
leftpos = glm::translate(leftpos, glm::vec3(0, -SCALE, 0));
leftpos = glm::scale(leftpos, glm::vec3(SCALE, 1, SCALE));
factory.constructWall(fw, sc, leftpos, glm::vec3(0, SCALE, SCALE),
        collChannel.GetAddress());
//Kuution oikea seinä
glm::mat4 rightpos;
rightpos = glm::rotate(rightpos, glm::radians(-90.0f), glm::vec3(0.0f,
        0.0f, 1.0f));
rightpos = glm::translate(rightpos, glm::vec3(0, -SCALE, 0));
rightpos = glm::scale(rightpos, glm::vec3(SCALE, 1, SCALE));
factory.constructWall(fw, sc, rightpos, glm::vec3(0, SCALE, SCALE),
        collChannel.GetAddress());
//Kuution takaseinä
glm::mat4 backpos;
backpos = glm::rotate(backpos, glm::radians(90.0f), glm::vec3(1.0f, 0.0f,
        0.0f));
backpos = glm::translate(backpos, glm::vec3(0, -SCALE, 0));
```

```
	backpos = glm::scale(backpos, glm::vec3(SCALE, 1, SCALE));
	factory.constructWall(fw, sc, backpos, glm::vec3(SCALE, SCALE, 0),
		collChannel.GetAddress());
	//Kuution etuseinä
	glm::mat4 frontpos;
	frontpos = glm::rotate(frontpos, glm::radians(-90.0f), glm::vec3(1.0f,
		0.0f, 0.0f));
	frontpos = glm::translate(frontpos, glm::vec3(0, -SCALE, 0));
	frontpos = glm::scale(frontpos, glm::vec3(SCALE, 1, SCALE));
	factory.constructWall(fw, sc, frontpos, glm::vec3(SCALE, SCALE, 0),
		collChannel.GetAddress());

	//Luodaan pallot
	for (int i = 0; i < SPHERES; i++){
		factory.constructRandomSphere(fw, sc, glm::vec3(SCALE - 1, SCALE -
			1, SCALE - 1), collChannel, WAIT);
	}

	//Käynnistetään pelisilmukka
	loop.start();

	//Odotetaan käyttäjän syötettä ennen lopettamista, jotta käyttäjä voi
	//lukea ruudulta suorituskykytiedot
	std::cin >> c;
	return 0;
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

Waiter.h:

```
#ifndef WAITER_H
#define WAITER_H

#define WAITER_ID 7

#include "stdafx.h"
#include <Component.h>

namespace stage{
	/**Pelioliokomponentti, joka simuloi raskasta laskentaa etsimällä tietyn
		määrän alkulukuja jokaisen ruudunpäivityksen aikana*/
	class Waiter : public Component{
	public:
		/**Luo uuden Waiter-komponentin.
		@param fw	Komponenttiaktoria hallinnoiva
				aktorijärjestelmä
		@param owner	Komponentin omistava peliolio
		@param limit	Se luku, johon asti alkulukuja etsitään joka
				ruudunpäivityksen aikana
		*/
		Waiter(Theron::Framework& fw, Theron::Address owner, int limit) :
			Component(fw, owner), limit(limit){}
		/**Suorittaa päivitysvaiheen laskennan
		@param up	Päivityspyyntöviesti
```

```
		@param from   Pyynnön lähettäjä
		*/
		virtual void update(const Update &up, Theron::Address from){
			for (int i = 3; i <= limit; i++){
				for (int j = 2; j < i; j++){
					if (i % j == 0) break;
				}
			}
			Send(AllDone(up.id), from);
		}
		/**Hakee Waiter-komponentin komponenttitunnuksen
		@returns	Tämän komponentin komponenttitunnus
		*/
		virtual int id(){ return WAITER_ID; }
	private:
		/**Se luku, johon asti alkulukuja etsitään jokaisen
		ruudunpäivityksen aikana*/
		int limit;
	};
}
#endif
```

Moduuli Stage_event:

ContextTracker.h:

```
#pragma once

#ifndef CONTEXTTRACKER_H
#define CONTEXTTRACKER_H

#include "stdafx.h"
#include "EventContext.h"
#include "CoreEvents.h"
#include <unordered_map>

namespace stage{
	/** Luokka, joka ylläpitää usean viestin käsittelyn ajan säilyviä
	konteksteja.*/
	class ContextTracker{
	public:
		/** Luo uuden kontekstilistan.
		@param fw	Se Theron::Framework, joka ylläpitää listan
				omistajaa
		@param owner	Listan omistajan Theron-osoite
		*/
		ContextTracker(Theron::Framework& fw, Theron::Address owner) :
			fw(fw), owner(owner), pending(){   }
		/** Luo uuden tapahtumakontekstin.
		@param oldID		Sen viestin tunnus, jonka seurauksena
					tämä konteksti luodaan
		@param newID		Tähän kontekstiin liittyvien viestien tunnus
		@param originalSender Alkuperäisen viestin lähettäjä
		@param responseCount Tähän viestiin odotettavien vastauksien määrä
					(oletus 1)
		@returns		Viite luotuun kontekstiolioon
		*/
		EventContext& addContext(uint64_t oldID, uint64_t newID,
			Theron::Address originalSender, int responseCount = 1);
		/** Hakee viitteen haluttuun kontekstiin
		HUOM: jos haluttua kontekstia ei ole, voi aiheuttaa virhetilanteen.
```

```
Tarkista tarvittaessa kontekstin olemassaolo contains()-metodilla.
@param id     Haettavan kontekstin tunnus
@returns      Viite haluttuun kontekstiin
*/
EventContext& getContext(uint64_t id);
/** Lisää halutun kontekstin vastausten määrää yhdellä
HUOM: jos haluttua kontekstia ei ole, voi aiheuttaa virhetilanteen.
Tarkista tarvittaessa kontekstin olemassaolo contains()-metodilla.
@param id     Sen kontekstin tunnus, jonka vastausten määrää
              kasvatetaan
*/
void increment(uint64_t id);
/** Muuttaa halutun kontekstin vastausten määrää
HUOM: jos haluttua kontekstia ei ole, voi aiheuttaa virhetilanteen.
Tarkista tarvittaessa kontekstin olemassaolo contains()-metodilla.
@param id            Sen kontekstin tunnus, jonka vastausten määrää
                     muutetaan
@param count  Uusi vastausmäärä
*/
void setResponseCount(uint64_t id, unsigned int count);
/** Vähentää halutun kontekstin vastausten määrää yhdellä
Jos tämä laskee vastausten määrän nollaan, kontekstin lopetusmetodi
suoritetaan ja konteksti poistetaan listasta
HUOM: jos haluttua kontekstia ei ole, voi aiheuttaa virhetilanteen.
Tarkista tarvittaessa kontekstin olemassaolo contains()-metodilla.
@param id     Sen kontekstin tunnus, jonka vastausten määrää
              lasketaan
*/
void decrement(uint64_t id);
/** Poistaa halutun kontekstin kontekstilistasta
@param id     Poistettavan kontekstin tunnus
*/
void remove(uint64_t id);
template <class MessageType>
/** Lähettää viestin, luo tarvittaessa sille uuden kontekstin ja
kasvattaa olemassaolevan kontekstin
odotettujen vastausten määrää yhdellä.
@param oldID                Sen viestin tunnus, jonka seurauksena
                            tämä konteksti luodaan
@param ev                   Lähetettävä viesti, oltava Event-luokan
                            aliluokka
@param recipient            Viestin vastaanottajan osoite
@param originalSender       Alkuperäisen viestin lähettäjä
@returns                    Viite luotuun kontekstiolioon
*/
void trackedSend(uint64_t oldID, const MessageType& ev,
       Theron::Address recipient, Theron::Address originalSender){
       if (!contains(ev.id)){
              addContext(oldID, ev.id, originalSender, 0);
       }
       pending[ev.id].responseCount++;
       fw.Send(ev, owner, recipient);
}
template <typename T>
/** Liittää kontekstiin muuttujan
@param id     Sen kontekstin tunnus, johon muuttuja
              liitetään
@param index  Muuttujan järjestysnumero (alkaa nollasta)
@param var    Muuttujan arvo
*/
void setVariable(uint64_t id, int index, T var){
```

```
                boost::any content = var;
                pending[id].setVar(index, content);
        }
        template <typename T>
        /** Hakee kontekstiin liitetyn muuttujan arvon
        @param id     Sen kontekstin tunnus, johon muuttuja on
                      liitetty
        @param index  Muuttujan järjestysnumero (alkaa nollasta)
        @returns      Muuttujan arvo
        */
        T getVariable(uint64_t id, int index){
                return boost::any_cast<T>(pending[id].getVar(index));
        }
        /** Generoi uuden viestitunnuksen
        @param returns       Viestin uniikisti määrittelevä tunnusarvo
        */
        uint64_t getNextID();
        /** Tarkistaa, onko halutulla avainarvolla tallennettu
        kontekstilistaan kontekstia
        @param id     Tutkittava avainarvo
        @returns      True, jos arvolla on tallennettu listaan konteksti
        */
        bool contains(uint64_t id);
    private:
        /** Tietorakenne, joka pitää kirjaa isäntäolion avoimista
        konteksteista*/
        std::unordered_map<uint64_t, EventContext> pending;
        /** Uniikkien viestitunnusten generointiin käytettävä laskuri*/
        uint32_t lastID = 0;
        /** Viite siihen Theron::Framework-olioin, joka hallinnoi tämän
        olion omistajaoliota*/
        Theron::Framework& fw;
        /** Tämän olion omistajaolion Theron-osoite*/
        Theron::Address owner;
    };
}
#endif
```

ContextTracker.cpp:

```
#include "stdafx.h"
#include "ContextTracker.h"

using namespace stage;

EventContext& ContextTracker::addContext(uint64_t oldID, uint64_t newID,
    Theron::Address originalSender, int responseCount){
    pending[newID] = EventContext(oldID, originalSender, responseCount);
    EventContext& context = pending[newID];
    //Asetetaan oletuskäsittelijä kontekstin loppuunsuorittamiselle:
    //palautetaan AllDone alkuperäiselle lähettäjälle
    context.finalize = [this, &context](){
        this->fw.Send(AllDone(context.getOriginalID()), this->owner,
            context.getOriginalSender());
    };
    context.error = context.finalize;
    return context;
}
EventContext& ContextTracker::getContext(uint64_t id){
    return pending[id];
}
```

```
void ContextTracker::increment(uint64_t id){
	EventContext& context = pending[id];
	context.responseCount++;
}
void ContextTracker::setResponseCount(uint64_t id, unsigned int count){
	EventContext& context = pending[id];
	context.responseCount = count;
}
void ContextTracker::decrement(uint64_t id){
	EventContext& context = pending[id];
	context.responseCount--;
	if (context.responseCount < 1){
		context.finalize();
		pending.erase(id);
	}
}
void ContextTracker::remove(uint64_t id){
	pending.erase(id);
}
uint64_t ContextTracker::getNextID(){
	lastID++;
	return Event::generateID(owner, lastID);
}
bool ContextTracker::contains(uint64_t id){
	if (pending.count(id)) return true;
	return false;
}
```

ContextVarList.h:

```
#ifndef CONTEXTVARLIST_H
#define CONTEXTVARLIST_H

#include "stdafx.h"
#include <boost\any.hpp>

namespace stage{
	/** Linkitetyn listan alkio, joka pitää sisällään tapahtumakontekstiin
	liittyvän muuttujan*/
	struct ContextVar{
		/** Muuttujan arvo*/
		boost::any content;
		/** Muuttujalistan seuraava alkio*/
		ContextVar* next = nullptr;
		/** Luo uuden lista-alkion*/
		ContextVar(boost::any content) : content(content){}
		/** Tuhoaa lista-alkion sekä rekursiivisesti kaikki sitä seuraavat
		alkiot*/
		~ContextVar(){
			if (next != nullptr) delete next;
		}
		/** Hakee rekursiivisesti muuttujalistasta yksittäisen muuttujan
		@param depth  Haettavan muuttujan järjestysluku tästä alkiosta
			      eteenpäin (0 palauttaa tämän alkion muuttujan)
		@returns		Haettavan muuttujan arvo
		*/
		boost::any get(int depth){
			if (depth < 1) return content;
			return next->get(depth - 1);
		}
		/** Asettaa rekursiivisesti muuttujalistaan yksittäisen muuttujan
```

```
                arvon
                @param depth  Asetettavan muuttujan järjestysluku tästä alkiosta
                              eteenpäin (0 asettaa tämän alkion muuttujan)
                @param content       Asetettavan muuttujan arvo
                */
                void set(int depth, boost::any content){
                        if (depth < 1) this->content = content;
                        else {
                                if (next == nullptr){
                                        boost::any a;
                                        next = new ContextVar(a);
                                }
                                next->set(depth - 1, content);
                        }
                }
        };
}
#endif
```

CoreEvents.h:

```
#ifndef COREEVENTS_H
#define COREEVENTS_H

#include "stdafx.h"

namespace stage {
        /** Tietue, jonka kaikki pelimoottorin viestit perivät.
        Sisältää viestikohtaisen viestitunnuksen.*/
        struct Event {
                /** Luo uuden viestin.
                @param originator    Viestin luojan Theron-osoite
                @param msgID         32-bittinen tunnus, joka yksilöi viestin
                                     muiden saman lähettäjän viestien joukossa
                */
                Event(Theron::Address originator, uint32_t msgID){
                        id = generateID(originator, msgID);
                }
                /** Luo uuden viestin.
                @param msgID  64-bittinen tunnus, joka yksilöi viestin kaikkien
                              pelimoottorin viestien joukossa
                */
                Event(uint64_t msgID){
                        id = msgID;
                }
                /** Viestin yksilöivä tunnusluku
                */
                uint64_t id;
                /** Apufunktio, joka muodostaa osoitteesta ja oliokohtaisesti
                uniikista 32-bittisestä tunnuksesta
                pelimoottorin sisällä yksilöllisen 64-bittisen tunnuksen
                @param originator    Tunnuksen luojan osoite
                @param msgID         Tunnuksen luojan lähettämien viestien joukossa
                                     yksilöllinen tunnusluku
                @returns             64-bittinen luku, joka yksilöi viestin
                                     pelimoottorin sisällä
                */
                static uint64_t generateID(Theron::Address originator, uint32_t
                        msgID){
                        uint64_t id = originator.AsInteger();
                        id = id << 32;
```

```
                id = id | msgID;
                return id;
        }
    };
    /** Viesti, jolla pyydetään pelioliota tai komponenttia päivittämään
    sisäinen tilansa uutta
    ruudunpäivitystä vastaavaksi*/
    struct Update : Event {
        /** Edellisestä ruudunpäivityksestä kulunut aika millisekunteina
        */
        float elapsedMS;
        Update(float ms, uint64_t msgID) : elapsedMS(ms), Event(msgID){}
        Update(float ms, Theron::Address originator, uint32_t msgID) :
        elapsedMS(ms), Event(originator, msgID){}
    };
    /** Viesti, jolla pyydetään pelioliota tai komponenttia piirtämään itsensä
    ruudulle*/
    struct Render : Event {
        Render(uint64_t msgID) : Event(msgID){}
        Render(Theron::Address originator, uint32_t msgID) :
            Event(originator, msgID){}
    };
    /** Viesti, joka ilmaisee pyydetyn aktorilaskennan päättyneen
    Viestin id-kenttä ilmaisee sen viestin, jonka käsittely on päättynyt*/
    struct AllDone : Event {
        AllDone(uint64_t msgID) : Event(msgID){}
    };
    /** Viesti, joka ilmaisee, että pyydetyssä aktorilaskennassa on kohdattu
    virhetilanne
    Viestin id-kenttä ilmaisee sen viestin, jonka käsittely on epäonnistunut*/
    struct Error : Event {
        Error(uint64_t msgID) : Event(msgID){}
    };
}
#endif
```

EventChannel.h:

```
#pragma once
#ifndef EVENTCHANNEL_H
#define EVENTCHANNEL_H

#include "stdafx.h"
#include <list>
#include "CoreEvents.h"
#include <iostream>
#include "LogActor.h"
#include "EventChannelManager.h"

namespace stage {
    /** Luokka, joka vastaa tapahtumaviestien lähettämisetä eteenpäin niistä
    kiinnostuneille aktoreille
    Ottaa vastaan viestit:
    <Tapahtumakanavan tyyppiparametri> (palauttaa AllDone)
    EventChannel::RegisterRecipient
    EventChannel::DeregisterRecipient
    EventChannelManager::ChannelMaintenance (palauttaa AllDone)
    */
    template <class MessageType>
    class EventChannel : public Theron::Actor{
    public:
```

```
	//---Viestit---
	/** Viesti, jolla lisätään uusi vastaaottaja kanavan
	vastaanottajalistaan*/
	struct RegisterRecipient {
		RegisterRecipient(Theron::Address &rec) : recipient(rec){}
		/** Rekisteröitävän aktorin Theron-osoite*/
		Theron::Address recipient;
	};
	/** Viesti, jolla poistetaan vastaanottaja vastaanottajalistasta*/
	struct DeregisterRecipient {
		DeregisterRecipient(Theron::Address &rec) : recipient(rec){}
		/** Poistettavan aktorin Theron-osoite
		*/
		Theron::Address recipient;
	};

	//---Metodit---

	/** Luo uuden tapahtumakanavan
	@param fw	Tapahtumakanavaa hallinnoiva Theron::Framework-olio
	*/
	EventChannel(Theron::Framework& fw) : Theron::Actor(fw),
		recipients(), tracker(fw, this->GetAddress()){
		RegisterHandler(this, &EventChannel<MessageType>::forward);
		RegisterHandler(this,
		&EventChannel<MessageType>::registerRecipient);
		RegisterHandler(this,
		&EventChannel<MessageType>::deregisterRecipient);
		RegisterHandler(this, &EventChannel<MessageType>::allDone);
		RegisterHandler(this, &EventChannel<MessageType>::error);
		RegisterHandler(this,
		&EventChannel<MessageType>::channelMaintenance);
	}
	/** Hakee listan kanavaa kuuntelemaan rekisteröityneistä aktoreista
	Säieturvallinen pelisilmukan päivitys- ja piirtovaiheissa
	@returns	Kanavaa kuuntelemaan rekisteröityneet aktorit
	*/
	const std::list<Theron::Address>& getRecipients(){
		return recipients;
	}
private:
	/** Lista aktoreista, jotka ovat rekisteröityneet kuuntelemaan
	kanavaa*/
	std::list<Theron::Address> recipients;
	/** Lista aktoreista, jotka lisätään kuuntelijalistaan
	ruudunpäivityksen lopussa*/
	std::list<Theron::Address> pendingAdd;
	/** Lista aktoreista, jotka poistetaan kuuntelijalistasta
	ruudunpäivityksen lopussa*/
	std::list<Theron::Address> pendingRemove;
	/** Tapahtumakanavan konteksteista kirjaa pitävä olio*/
	ContextTracker tracker;

	/** Lähettää viestin eteenpäin kaikille tapahtumakanavaan
	rekisteröityneille aktoreille
	alkuperäistä lähettäjää lukuun ottamatta
	@param msg	Viesti, joka jaetaan kaikille kuuntelijoille
	@param from	Viestin lähettäjä
	*/
	void forward(const MessageType &msg, const Theron::Address from){
		int sent = 0;
```

```
		for (std::list<Theron::Address>::const_iterator it =
			recipients.begin(); it != recipients.end(); it++){
			if (*it != from){
				tracker.trackedSend(msg.id, msg, *it, from);
				sent++;
			}
		}
		if (sent == 0){
			//Jos ei kuuntelijoita, kaikki valmista
			Send(AllDone(msg.id), from);
		}
}
/** Liittää uuden kuuntelijan kuuntelijalistaan ruudunpäivityksen
lopussa
@param msg	Rekisteröintipyyntö
@param from	Pyynnön lähettäjä
*/
void registerRecipient(const RegisterRecipient& msg, const
		Theron::Address from){
		pendingAdd.push_back(msg.recipient);
}
/** Poistaa kuuntelijan kuuntelijalistasta ruudunpäivityksen
lopussa
@param msg	Poistopyyntö
@param from	Pyynnön lähettäjä
*/
void deregisterRecipient(const DeregisterRecipient& msg, const
Theron::Address from){
		pendingRemove.push_back(msg.recipient);
}
/** Lisää ja poistaa kuuntelijalistasta kaikki lisäystä tai poistoa
odottavat aktorit
@param msg	Kanavanhuoltopyyntö
@param from	Pyynnön lähettäjä
*/
void channelMaintenance(const
		EventChannelManager::ChannelMaintenance& msg, const
		Theron::Address from){
		for (std::list<Theron::Address>::const_iterator it =
			pendingAdd.begin(); it != pendingAdd.end(); it++){
			recipients.push_back(*it);
		}
		pendingAdd.clear();
		for (std::list<Theron::Address>::const_iterator it1 =
			pendingRemove.begin(); it1 != pendingRemove.end();
			it1++){
			for (std::list<Theron::Address>::const_iterator it2 =
				recipients.begin(); it2 != recipients.end();
				it2++){
				if (*it1 == *it2) recipients.erase(it2);
			}
		}
		pendingRemove.clear();
		Send(AllDone(msg.id), from);
}
/**Käsittelee eteenpäin lähetettyjen viestien käsittelyn
päättymisestä ilmoittavat viestit
@param msg	Valmistumisesta ilmoittava viesti
@param from	Viestin lähettäjä
*/
void allDone(const AllDone& msg, const Theron::Address from){
```

```
                if (tracker.contains(msg.id)) tracker.decrement(msg.id);
        }
        /**Käsittelee eteenpäin lähetettyjen viestien käsittelyssä
        tapahtuneesta virheestä ilmoittavat viestit
        @param msg    Virheestä ilmoittava viesti
        @param from   Viestin lähettäjä
        */
        void error(const Error &msg, Theron::Address from){
                LOGERR(std::string("Warning: component ") + from.AsString()
                        + " reported error during processing");
                if (tracker.contains(msg.id)) tracker.decrement(msg.id);
        }
    };
}
#endif
```

EventChannelManager.h:

```
#ifndef EVENTCHANNELMANAGER_H
#define EVENTCHANNELMANAGER_H

#include "stdafx.h"
#include "ContextTracker.h"
#include "CoreEvents.h"

namespace stage{
    /** Luokka, joka pitää kirjaa tapahtumakanavista ja huoltaa niitä*/
    class EventChannelManager : public Theron::Actor{
    public:
        //---Viestit---

        /** Viesti, jolla pyydetään tapahtumakanavia suorittamaan huollon
        eli lisäämään ja
        poistamaan lisäystä tai poistoa odottavat aktorit
        vastaanottajalistastaan*/
        struct ChannelMaintenance: public Event{
                ChannelMaintenance(uint64_t id): Event(id){}
        };

        //---Metodit---

        /** Luo uuden EventChannelManager-aktorin
        @param fw     Aktoria hallinnoiva Theron::Framework
        */
        EventChannelManager(Theron::Framework& fw): Theron::Actor(fw),
                tracker(fw, this->GetAddress()){
                RegisterHandler(this,
                &EventChannelManager::channelMaintenance);
                RegisterHandler(this, &EventChannelManager::allDone);
        }
        /** Lisää kanavan huollettavien kanavien listaan
        @param channel       Lisättävä kanava
        */
        void addChannel(Theron::Address channel){
                channels.push_back(channel);
        }
    private:
        /** Lista hallinnoitavista kanavista*/
        std::list<Theron::Address> channels;
        /** Tapahtumakonteksteista kirjaa pitävä olio*/
        ContextTracker tracker;
```

```
		/** Pyytää kaikkia hallinnoimiaan kanavia suorittamaan huollon
		@param msg		Huoltopyyntö
		@param sender Pyynnön lähettäjä
		*/
		void channelMaintenance(const ChannelMaintenance& msg,
			Theron::Address sender){
			if (channels.size() == 0) Send(AllDone(msg.id), sender);
			ChannelMaintenance newmsg(tracker.getNextID());
			for (std::list<Theron::Address>::const_iterator it =
				channels.begin(); it != channels.end(); it++){
				tracker.trackedSend(msg.id, newmsg, *it, sender);
			}
		}
		/** Käsittelee kanavilta saapuvat huollon päättymisestä ilmoittavat
		viestit
		@param msg		Huollon päättymisestä kertova viesti
		@param sender Viestin lähettäjä
		*/
		void allDone(const AllDone& msg, Theron::Address sender){
			if (tracker.contains(msg.id)) tracker.decrement(msg.id);
		}
	};
}
#endif
```

EventContext.h:

```
#ifndef EVENTCONTEXT_H
#define EVENTCONTEXT_H

#include "stdafx.h"
#include <functional>
#include <iostream>
#include "ContextVarList.h"

namespace stage{
	/** Luokka, joka ylläpitää kontekstitietoja usean viestin käsittelyn yli.
	Mahdollistaa "paikallisten" muuttujien käsittelemisen useasta toisiinsa
	liittyvästä viestinkäsittelijästä.*/
	class EventContext{
	public:
		//---Kentät---

		/** Tähän kontekstiin liittyvien vastausviestien lukumäärä.
		Esim. jos kontekstin yhteydessä lähetetään viestit kolmelle
		aktorille, responseCount:n arvoksi asetetaan 3,
		jotta lopetusmetodi suoritetaan, kun kaikki ovat vastanneet*/
		unsigned int responseCount;
		/** Kontekstin lopetusmetodi. Suoritetaan automaattisesti, kun
		kaikki odotetut vastaukset ovat saapuneet.*/
		std::function<void()> finalize;
		/** Kontekstin virhemetodi. Suoritetaan, jos kontekstiin liittyen
		lähetettyjen viestien käsittelyssä tapahtuu virhe.*/
		std::function<void()> error;
		/** Osoitin ensimmäiseen kontekstimuuttujaan*/
		ContextVar* varHead = nullptr;

		//---Metodit---

		/** Luo uuden tapahtumakontekstin.
```

```
		@param id			Sen viestin tunnusluku, jonka vuoksi tämä
						konteksti luotiin
		@param sender		Sen viestin lähettäjä, jonka vuoksi tämä
						konteksti luotiin
		@param responseCount Tähän viestiin odotettujen vastausten määrä
		*/
		EventContext(uint64_t id, Theron::Address sender, unsigned int
			responseCount): originalID(id), originalSender(sender),
			responseCount(responseCount){}
		/** Luo tyhjän tapahtumakontekstin kontekstilistan tietorakenteiden
			initialisointia varten*/
		EventContext() : originalID(0), originalSender(0), responseCount(0)
			{
			//Estetään lopetusmetodin kutsuminen tyhjälle kontekstille
			finalize = [](){abort(); };
			error = finalize;
		}
		/** Estetään kopiokonstruktori*/
		EventContext(EventContext& other) = delete;
		/** Tuhoaa yksittäisen tapahtumakontekstin ja siihen liittyvät
		muuttujat*/
		~EventContext(){
			if (varHead != nullptr){
				delete varHead;
			}
		}
		/** Hakee kontekstiin liitetyn muuttujan arvon
		@param depth  Muuttujan järjestysnumero (alkaa nollasta)
		@returns		Muuttujan arvo
		*/
		boost::any getVar(int depth) const{
			return varHead->get(depth);
		}
		/** Liittää kontekstiin muuttujan
		@param depth  Muuttujan järjestysnumero (alkaa nollasta)
		@param var		Muuttujan arvo
		*/
		void setVar(int depth, boost::any var){
			if (varHead == nullptr){
				boost::any a = 0;
				varHead = new ContextVar(a);
			}
			varHead->set(depth, var);
		}
		/** Palauttaa sen viestin tunnuksen, joka aiheutti tämän kontekstin
		luomisen
		@returns	alkuperäinen viestitunnus
		*/
		uint64_t getOriginalID() const { return originalID; }
		/** Palauttaa sen viestin lähettäjän, joka aiheutti tämän
		kontekstin luomisen
		@returns	alkuperäisen lähettäjän Theron-osoite
		*/
		Theron::Address getOriginalSender() const { return
			originalSender; }
	private:
		/** Sen viestin tunnus, joka aiheutti tämän kontekstin luomisen*/
		uint64_t originalID;
		/** Sen viestin lähettäjä, joka aiheutti tämän kontekstin
		luomisen*/
		Theron::Address originalSender;
```

```
	};
}
#endif
```

LogActor.h:

```
#ifndef LOGACTOR_H
#define LOGACTOR_H

#include "stdafx.h"
#include <Logger.h>
#include <iostream>

/**Makro, jolla kirjoitetaan lokiin normaali lokiviesti*/
#define LOGMSG(MSG) Send(LogActor::LogMessage(MSG), LogActor::getGlobalLogger());
/**Makro, jolla kirjoitetaan lokiin normaali virheilmoitus*/
#define LOGERR(MSG) Send(LogActor::LogError(MSG), LogActor::getGlobalLogger());

namespace stage{
	/** Aktori, joka tarjoaa pelimoottorille lokipalvelun.
	Kapseloi lokipalvelun aktorin sisään, jotta lokiin voidaan kirjoittaa
	säieturvallisesti
	Ensimmäinen luotu lokiaktori toimii globaalina singletonina, jota muut
	aktorit oletuksena hyödyntävät.
	Ottaa vastaan viestit:
	LogActor::LogMessage
	LogActor::LogError
	LogActor::Terminate (palauttaa LogActor::Terminate)
	*/
	class LogActor : public Theron::Actor {
	public:
		//---Viestit---

		/** Viesti, jolla pyydetään lokiaktoria kirjoittamaan merkintä
		lokiin*/
		struct LogMessage{
			/** Lokiin kirjoitettava viesti*/
			std::string message;
			LogMessage(std::string msg) : message(msg){}
		};
		/** Viesti, jolla pyydetään lokiaktoria kirjoittamaan merkintä
		virhelokiin*/
		struct LogError{
			/** Lokiin kirjoitettava viesti*/
			std::string message;
			LogError(std::string msg) : message(msg){}
		};
		/** Viesti, jolla pyydetään lokia valmistautumaan järjestelmän
		sammuttamiseen.
		Kun kaikki nykyiset viestit on saatu käsiteltyä, lokiaktori ei enää
		ota vastaan viestejä ja palautaa takaisin
		saman Terminate-viestin, joka ilmaisee, että aktori voidaan tuhota
		turvallisesti.*/
		struct Terminate{};

		//---Metodit---

		/** Luo uuden lokiaktorin. Jos muita lokiaktoreita ei ole luotu,
		luotavan aktorin
		Theron-osoitteen saa selville getGlobalLogger-funktiolla.
		@param fw		Lokiaktoria hallinnoiva Theron::Framework
```

```
		@param standard		ostream, johon kirjoitetaan normaalit
					lokiviestit (oletusarvo cout)
		@param error		ostream, johon kirjoitetaan virheilmoitukset
					(oletusarvo cerr)
		*/
		LogActor(Theron::Framework& fw, std::ostream& standard = std::cout,
			std::ostream& error = std::cerr);
		/** Palauttaa globaalin lokiaktorin osoitteen.
		Globaali lokiaktori toimii oletuslokina, johon kirjoitetaan kaikki
		lokiviestit ellei toisin määritellä
		@returns	Globaalin lokiaktorin Theron-osoite
		*/
		static Theron::Address getGlobalLogger(){ return globalLogger; }
	private:
		/** Globaalin lokiaktorin Theron-osoite*/
		static Theron::Address globalLogger;
		/** Tätä aktoria sammumaan pyytäneen aktorin Theron-osoite*/
		Theron::Address terminator;
		/** Lokiolio, joka kirjoittaa vastaanotettavat viestit lokiin*/
		stage_common::Logger logger;

		/** Kirjoittaa merkinnän lokiin
		@param msg	LogMessage-tietue, joka sisältää
				kirjoitettavan viestin
		@param sender Lokimerkintää pyytävän aktorin osoite
		*/
		void logMessage(const LogMessage& msg, Theron::Address sender);
		/** Kirjoittaa virheilmoituksen lokiin
		@param msg	LogMessage-tietue, joka sisältää
				kirjoitettavan viestin
		@param sender Lokimerkintää pyytävän aktorin osoite
		*/
		void logError(const LogError& msg, Theron::Address sender);
		/** Asettaa lokiaktorin tilaan, jossa se sammuu, kun se on saanut
		kaikki lokimerkintänsä kirjoitettua lokiin
		@param msg		Sammutuspyyntö
		@param sender Sammutusta pyytävän aktorin osoite
		*/
		void terminate(const Terminate& msg, Theron::Address sender);
		/** Tarkistaa, onko vielä kirjoitettavia viestejä - jos ei,
		sammuttaa lokiaktorin
		*/
		void tryShutdown();
	};
}
#endif
```

LogActor.cpp:

```
#include "stdafx.h"
#include "LogActor.h"

using namespace stage;

/** Globaalin lokiosoitteen alkuarvo*/
Theron::Address LogActor::globalLogger = Theron::Address::Null();

LogActor::LogActor(Theron::Framework& fw, std::ostream& standard, std::ostream&
	error) : Theron::Actor(fw), logger(standard, error){
	if (globalLogger == Theron::Address::Null()) globalLogger = this
		->GetAddress();
```

```
	terminator = Theron::Address::Null();
	RegisterHandler(this, &LogActor::logMessage);
	RegisterHandler(this, &LogActor::logError);
	RegisterHandler(this, &LogActor::terminate);
}
void LogActor::logMessage(const LogMessage& msg, Theron::Address sender){
	logger.Log(msg.message);
	if (terminator != Theron::Address::Null()){
		tryShutdown();
	}
}
void LogActor::logError(const LogError& msg, Theron::Address sender){
	logger.LogError(msg.message);
	if (terminator != Theron::Address::Null()){
		tryShutdown();
	}
}
void LogActor::terminate(const Terminate& msg, Theron::Address sender){
	logger.Log("Logger ordered to shut down");
	this->terminator = sender;
	tryShutdown();
}
void LogActor::tryShutdown(){
	if (this->GetNumQueuedMessages() <= 1){
		logger.Log("Logger shutting down");
		DeregisterHandler(this, &LogActor::logMessage);
		DeregisterHandler(this, &LogActor::logError);
		DeregisterHandler(this, &LogActor::terminate);
		Send(Terminate(), terminator);
	}
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN

#include <Theron\Theron.h>
```

Moduuli Stage_scene:

Component.h:

```
#pragma once
#ifndef COMPONENT_H
#define COMPONENT_H

#include "stdafx.h"
#include "GameObject.h"
#include <CoreEvents.h>
#include <ContextTracker.h>
#include <iostream>

namespace stage{
	/** Abstrakti pelimoottorikomponenttien yliluokka.
	Kaikki pelimoottoreihin liitettävät komponentit perivät tämän luokan.
	Ottaa vastaan viestit:
	Update (palauttaa AllDone)
```

```
Render (palauttaa AllDone)
Component::GetComponentID (palauttaa Component::ComponentID)
GameObject::GetComponent (palauttaa GameObject::ComponentFound tai
	AllDone)
*/
class Component : public Theron::Actor{
public:
	/** Viesti, joka kertoo lähettäjänsä tyyppitunnuksen*/
	struct ComponentID : Event{
		/** Lähettäjän tyyppitunnus
		*/
		unsigned int compID;
		ComponentID(uint64_t id, int compID) : Event(id),
			compID(compID){}
	};
	/** Viesti, jolla pyydetään komponenttia lähettämään
	tyyppitunnuksensa*/
	struct GetComponentID : Event{
		GetComponentID(uint64_t id) : Event(id){}
	};
	/** Luo uuden pelimoottorikomponentin.
	HUOM: luo komponenttiolio aina new:llä äläkä tuhoa sitä itse.
	Komponenttiolio tuhotaan aina automaattisesti, kun sen omistava
	peliolio tuhotaan.
	@param fw		Komponenttia hallinnoiva Theron::Framework
	@param owner  Komponentin omistava peliolio
	*/
	Component(Theron::Framework &fw, Theron::Address owner);
protected:
	/**	Komponentin omistava peliolio*/
	Theron::Address owner;
	/** Komponentin tapahtumakontekstien tilaa ylläpitävä olio*/
	ContextTracker tracker;

	/** Päivittää komponentin tilan.
	Jos tämä metodi ylikirjoitetaan aliluokassa, komponentti suorittaa
	laskentaa
	pelisilmukan päivitysvaiheessa.
	@param up	Päivityspyyntö
	@param from	Pyynnön lähettäjä
	*/
	virtual void update(const Update &up, Theron::Address from);
	/** Piirtää komponentin ruudulle.
	Jos tämä metodi ylikirjoitetaan aliluokassa, komponentti suorittaa
	laskentaa
	pelisilmukan piirtovaiheessa.
	@param up	Piirtopyyntö
	@param from	Pyynnön lähettäjä
	*/
	virtual void render(const Render &rend, Theron::Address from);
	/** Oletuskäsittelijä AllDone-viesteille (laskee viestiin liittyvän
	kontekstin
	odotettujen vastausten määrää yhdellä)
	Voidaan ylikirjoittaa aliluokassa.
	@param msg	AllDone-viesti
	@param from	Viestin lähettäjä
	*/
	virtual void allDone(const AllDone& msg, Theron::Address from);
	/** Oletuskäsittelijä Error-viesteille (kutsuu viestiin liittyvän
	kontekstin
	Error-metodia ja poistaa kontekstin)
```

```
		Voidaan ylikirjoittaa aliluokassa.
		@param msg	Error-viesti
		@param from	Viestin lähettäjä
		*/
		virtual void error(const Error& msg, Theron::Address from);
		/** Abstrakti metodi, joka palauttaa komponentin tyyppitunnuksen
		@returns	Komponentin tyypistä riippuva tunnusluku
		*/
		virtual int id() = 0;
	private:
		/** Kysyy komponentilta, onko se tiettyä tyyppiä
		@param msg	Etsityn tyypin tyyppitunnuksen sisältävä viesti
		@param from	Viestin lähettäjä
		Vastaa viestillä GameObject::ComponentFound jos tämä komponentti on
		etsittyä tyyppiä,
		muutoin viestillä AllDone
		*/
		void isType(const GameObject::GetComponent &msg, Theron::Address
			from);
		/** Kysyy komponentilta sen tyyppitunnusta
		@param msg	Pyyntöviesti
		@param from	Pyynnön lähettäjä
		*/
		void getId(const GetComponentID &msg, Theron::Address from);
	};
}
#endif
```

Component.cpp:

```
#include "stdafx.h"
#include "Component.h"

using namespace stage;

Component::Component(Theron::Framework &fw, Theron::Address owner) :
	Theron::Actor(fw), owner(owner), tracker(fw, this->GetAddress()){
	RegisterHandler(this, &Component::update);
	RegisterHandler(this, &Component::render);
	RegisterHandler(this, &Component::isType);
	RegisterHandler(this, &Component::allDone);
	RegisterHandler(this, &Component::error);
	Send(GameObject::AddComponent(this, tracker.getNextID()), owner);
	RegisterHandler(this, &Component::getId);
}
void Component::update(const Update &up, Theron::Address from){
	Send(AllDone(up.id), from);
}
void Component::render(const Render &rend, Theron::Address from){
	Send(AllDone(rend.id), from);
}
void Component::isType(const GameObject::GetComponent &msg, Theron::Address from)
{
	if (msg.compID == id()){
		Send(GameObject::ComponentFound(msg.id, this->GetAddress()), from);
	}
	else Send(AllDone(msg.id), from);
}
void Component::getId(const GetComponentID &msg, Theron::Address from){
	Send(ComponentID(msg.id, id()), from);
}
```

```
void Component::allDone(const AllDone& msg, Theron::Address from){
	if (tracker.contains(msg.id)) tracker.decrement(msg.id);
}
void Component::error(const Error& msg, Theron::Address from){
	if (tracker.contains(msg.id)){
		tracker.getContext(msg.id).error();
		tracker.remove(msg.id);
	}
}
```

GameObject.h:

```
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include "stdafx.h"
#include <Theron\Theron.h>
#include <string>
#include <list>
#include <CoreEvents.h>
#include <iostream>
#include <ContextTracker.h>

namespace stage{
	class Component;
	/** Luokka, joka mallintaa yksittäisen peliolion, joka kokoaa yhteen
	joukon toisiinsa liittyviä komponentteja
	Ottaa vastaan viestit:
	Update (palauttaa AllDone)
	Render (palauttaa AllDone)
	GameObject::AddComponent (ei palauta vastausta)
	GameObject::GetComponent (palauttaa GameObject::ComponentFound tai Error)
	*/
	class GameObject : public Theron::Actor{
	public:
		//---Viestit---

		/** Viesti, joka pyytää pelioliota liittämään itseensä komponentin
		Liitetty komponentti siirtyy peliolion hallintaan ja tuhotaan
		automaattisesti, kun peliolio tuhoutuu*/
		struct AddComponent : Event {
			/** Osoitin liitettävään komponenttiolioon*/
			Component* component;
			AddComponent(Component* component, uint64_t id) :
				component(component), Event(id){}
		};
		/** Hakee peliolion komponenttilistasta tietyntyyppiset
		komponenttioliot*/
		struct GetComponent : Event{
			/** Haettavien komponenttien komponenttitunnus*/
			int compID;
			GetComponent(uint64_t id, int compID) : Event(id),
				compID(compID){}
		};
		/** Viesti, joka kertoo GetComponent-viestin käsittelyssä löytyneen
		komponentin osoitteen*/
		struct ComponentFound : Event{
			/** Löytyneen komponentin Theron-osoite*/
			Theron::Address component;
			ComponentFound(uint64_t id, Theron::Address component) :
			Event(id), component(component){}
```

```
		};

		//---Metodit---

		/** Luo uuden tyhjän peliolion
		@param fw	Pelioliota hallinnoiva Theron::Framework
		*/
		GameObject(Theron::Framework &fw);
		/** Tuhoaa peliolion ja kaikki sen komponentit*/
		~GameObject();
	private:
		/** Peliolion kaikki komponentit sisältävä lista*/
		std::list<Component*> components;
		/** Peliolion viestikontekstilista*/
		ContextTracker tracker;

		/** Pyytää kaikkia peliolion komponentteja päivittämään tilansa
		@param msg	Päivityspyyntö
		@param from	Pyynnön lähettäjä
		*/
		void update(const Update &msg, Theron::Address from);
		/** Pyytää kaikkia peliolion komponentteja piirtämään itsensä
		ruudulle
		@param msg	Piirtopyyntö
		@param from	Pyynnön lähettäjä
		*/
		void render(const Render &msg, Theron::Address from);
		/** Liittää peliolioon uuden komponentin
		@param msg	Liitettävän komponentin määrittelevä viesti
		@param from	Komponentin lisäyspyynnön lähettäjä
		*/
		void addComponent(const AddComponent &msg, Theron::Address from);
		/** Hakee peliolioon liitetyt tiettyä tyyppiä olevat komponentit
		@param msg	Haettavan komponenttityypin määrittelevä viesti
		@param from	Hakupyynnön lähettäjä
		*/
		void getComponent(const GetComponent &msg, Theron::Address from);
		/** Palauttaa getComponent-funktion löytämän komponentin osoitteen
		alkuperäiselle pyytäjälle
		@param msg	Komponentin löytymisestä kertova viesti
		@param from	Komponentin löytymisestä kertovan aktorin osoite
		*/
		void componentFound(const ComponentFound &msg, Theron::Address
			from);
		/** Pitää kirjaa komponenttien laskennan edistymisestä
		@param msg	Ilmoitus komponentin laskennan onnistumisesta
		@param from	Ilmoituksen lähettäjä
		*/
		void allDone(const AllDone &msg, Theron::Address from);
		/** Käsittelee komponenttien laskennassa tapahtuneet virheet
		@param msg	Ilmoitus komponentin laskennan epäonnistumisesta
		@param from	Ilmoituksen lähettäjä
		*/
		void error(const Error &msg, Theron::Address from);
	};
}
#endif
```

GameObject.cpp:

```
#include "stdafx.h"
```

```
#include "GameObject.h"
#include "Component.h"
#include <boost\any.hpp>
#include <iostream>
#include <LogActor.h>

using namespace stage;
typedef std::list<Component*>::iterator comp_iterator;

GameObject::GameObject(Theron::Framework &fw) : Theron::Actor(fw), tracker(fw,
	this->GetAddress()) {
	RegisterHandler(this, &GameObject::update);
	RegisterHandler(this, &GameObject::render);
	RegisterHandler(this, &GameObject::addComponent);
	RegisterHandler(this, &GameObject::getComponent);
	RegisterHandler(this, &GameObject::allDone);
	RegisterHandler(this, &GameObject::error);
	RegisterHandler(this, &GameObject::componentFound);
}
GameObject::~GameObject(){
	for (comp_iterator i = components.begin(); i != components.end(); i++){
		delete *i;
	}
}
void GameObject::addComponent(const AddComponent &msg, Theron::Address from){
	components.push_back(msg.component);
}
void GameObject::getComponent(const GetComponent &msg, Theron::Address from){
	uint64_t id = tracker.getNextID();
	GetComponent newMsg(id, msg.compID);
	for (comp_iterator i = components.begin(); i != components.end(); i++){
		tracker.trackedSend<GetComponent>(msg.id, newMsg, (*i)
			->GetAddress(), from);
	}
	tracker.setVariable<bool>(id, 0, false);
	EventContext& ev = tracker.getContext(id);
	ev.finalize = [this, &ev](){
		bool result = boost::any_cast<bool>(ev.getVar(0));
		if (!result){
			this->GetFramework().Send(Error(ev.getOriginalID()), this
			->GetAddress(), ev.getOriginalSender());
		}
	};
}
void GameObject::componentFound(const ComponentFound &msg, Theron::Address from){
	if (tracker.contains(msg.id)){
		Send(ComponentFound(tracker.getContext(msg.id).getOriginalID(),
			msg.component),
			tracker.getContext(msg.id).getOriginalSender());
		tracker.setVariable<bool>(msg.id, 0, true);
		tracker.decrement(msg.id);
	}
}
void GameObject::update(const Update &msg, Theron::Address from){
	if (components.size() == 0){
		Send(AllDone(msg.id), from);
	}
	else {
		uint64_t id = tracker.getNextID();
		for (comp_iterator i = components.begin();i != components.end();
			i++){
```

```
                        tracker.trackedSend<Update>(msg.id, Update(msg.elapsedMS,
                                id), (*i)->GetAddress(), from);
                }
        }
}
void GameObject::render(const Render &msg, Theron::Address from){
        if (components.size() == 0){
                Send(AllDone(msg.id), from);
        }
        else {
                uint64_t id = tracker.getNextID();
                for (comp_iterator i = components.begin(); i != components.end();
                        i++){
                        tracker.trackedSend<Render>(msg.id, Render(id), (*i)
                                ->GetAddress(), from);
                }
        }
}
void GameObject::allDone(const AllDone &msg, Theron::Address from){
        if (tracker.contains(msg.id)) tracker.decrement(msg.id);
}
void GameObject::error(const Error &msg, Theron::Address from){
        LOGERR(std::string("Warning: component ") + from.AsString() + " reported
                error during processing");
        if (tracker.contains(msg.id)) tracker.decrement(msg.id);
}
```

Scene.h:

```
#ifndef SCENE_H
#define SCENE_H

#include "stdafx.h"
#include "GameObject.h"
#include <EventChannel.h>
#include <CoreEvents.h>
#include <iostream>

namespace stage {
        /** Luokka, joka mallintaa pelimaailman alueen.
        Ottaa vastaan viestit:
        Update (palauttaa AllDone)
        Render (palauttaa AllDone)
        Scene::CreateObject (palauttaa Scene::NewObject)
        */
        class Scene : public Theron::Actor{
                friend class SceneManager;
        public:
                /** Viesti, jolla pyydetään luomaan pelimaailmaan uusi peliolio*/
                struct CreateObject : Event{
                        CreateObject(uint64_t id) : Event(id){}
                };
                /** Viesti, joka palauttaa luodun peliolion osoitteen*/
                struct NewObject : Event{
                        NewObject(uint64_t id, Theron::Address obj) : Event(id),
                                object(obj){}
                        /** Uuden olion osoite
                        */
                        Theron::Address object;
                };
                /** Tuhoaa pelialueen ja kaikki sen sisältämät pelioliot
```

```
		*/
		~Scene();
	private:
		/** Lista pelialueen sisältämistä peliolioista*/
		std::list<GameObject*> objects;
		/** Pelialueen lähettämien viestien kontekstilista*/
		ContextTracker tracker;

		/** Luo uuden pelialueen
		@param fw	Pelialuetta hallinnoiva Theron::Framework
		*/
		Scene(Theron::Framework &fw);
		/** Luo pelialueelle uuden peliolion
		@param msg		Peliolion luontia pyytävä viesti
		@param sender Viestin lähettäjä
		*/
		void createObject(const CreateObject &msg, Theron::Address sender);
		/** Päivittää pelialueen tilan pyytämällä sen peliolioita
		päivittämään tilansa
		@param msg		Päivityspyyntö
		@param sender Pyynnön lähettäjä
		*/
		void update(const Update &msg, Theron::Address sender);
		/** Piirtää pelialueen ruudulle pyytämällä sen peliolioita
		piirtämään itsensä
		@param msg		Piirtopyyntö
		@param sender Pyynnön lähettäjä
		*/
		void render(const Render &msg, Theron::Address sender);
		/** Pitää kirjaa peliolioiden laskennan edistymisestä
		@param msg	Ilmoitus peliolion laskennan onnistumisesta
		@param from	Ilmoituksen lähettäjä
		*/
		void allDone(const AllDone &msg, Theron::Address sender);
		/** Käsittelee peliolioiden laskennassa tapahtuneet virheet
		@param msg	Ilmoitus peliolion laskennan epäonnistumisesta
		@param from	Ilmoituksen lähettäjä
		*/
		void error(const Error &msg, Theron::Address sender);
	};
}
#endif
```

Scene.cpp:

```
#include "stdafx.h"
#include "Scene.h"
#include <iostream>
#include <LogActor.h>

using namespace stage;

Scene::Scene(Theron::Framework &fw) : Theron::Actor(fw), tracker(fw, this
	->GetAddress()){
	RegisterHandler(this, &Scene::update);
	RegisterHandler(this, &Scene::render);
	RegisterHandler(this, &Scene::allDone);
	RegisterHandler(this, &Scene::error);
	RegisterHandler(this, &Scene::createObject);
}
void Scene::createObject(const Scene::CreateObject &msg, Theron::Address sender){
```

```
	GameObject* newObject = new GameObject(this->GetFramework());
	objects.push_back(newObject);
	Send(NewObject(msg.id, newObject->GetAddress()), sender);
}
Scene::~Scene(){
	for (std::list<GameObject*>::iterator i = objects.begin(); i !=
		objects.end(); i++){
		delete *i;
	}
}
void Scene::update(const Update &msg, Theron::Address sender){
	uint64_t id = tracker.getNextID();
	for (std::list<GameObject*>::iterator it = objects.begin(); it !=
		objects.end(); it++){
		tracker.trackedSend<Update>(msg.id, Update(msg.elapsedMS, id),
		(*it)->GetAddress(), sender);
	}
}
void Scene::render(const Render &msg, Theron::Address sender){
	uint64_t id = tracker.getNextID();
	for (std::list<GameObject*>::iterator it = objects.begin(); it !=
		objects.end(); it++){
		tracker.trackedSend<Render>(msg.id, Render(id), (*it)
		->GetAddress(), sender);
	}
}
void Scene::allDone(const AllDone &msg, Theron::Address sender){
	if (tracker.contains(msg.id)) tracker.decrement(msg.id);
}
void Scene::error(const Error &msg, Theron::Address sender){
	LOGERR(std::string("Warning: game object ") + sender.AsString() + "
		reported error during processing");
	if (tracker.contains(msg.id)) tracker.decrement(msg.id);
}
```

SceneManager.h:

```
#ifndef SCENEMANAGER_H
#define SCENEMANAGER_H

#include "stdafx.h"
#include <vector>
#include "Scene.h"

namespace stage{
	/** Abstrakti luokkka, joka tarjoaa aktorirajapinnan pääsäikeessä
	suoritettavaan pelisilmukkaan
	Ottaa vastaan viestit:
	SceneManager::Abort (ei lähetä vastausta)
	SceneManager::SetActiveScene (vastaa AllDone)
	SceneManager::CreateScene (vastaa SceneManager::NewScene)
	*/
	class SceneManager{
	public:
		//---Viestit---

		/** Viesti, joka ilmoittaa, että pelin suoritus on lopetettava*/
		struct Abort{};
		/** Viesti, jolla pyydetään pelimoottoria vaihtamaan aktiivista
		pelialuetta*/
		struct SetActiveScene : Event{
```

```
		/** Uuden pelialueen tunnusnumero*/
		unsigned int scene;
		SetActiveScene(uint64_t id, unsigned int scene) : Event(id),
			scene(scene){}
	};
	/** Viesti, jolla pyydetään pelimoottoria luomaan uusi pelialue
	*/
	struct CreateScene : Event{
		CreateScene(uint64_t id) : Event(id){}
	};
	/** Vastaus pelialueen luontipyyntöön - ilmoittaa uuden pelialueen
	tunnusnumeron ja osoitteen
	*/
	struct NewScene : Event{
		/** Uuden pelialueen tunnusnumero*/
		unsigned int scene;
		/** Uuden pelialueen Theron-osoite*/
		Theron::Address scAddress;
		NewScene(uint64_t id, unsigned int scene, Theron::Address
			scAddress) : Event(id), scene(scene),
			scAddress(scAddress){}
	};

	//---Metodit---

	/** Luo uuden pelialueiden hallintaolion*/
	SceneManager();
	/** Tuhoaa pelialueiden hallintaolion
	Toteutus:
	for (std::vector<Scene*>::iterator i = scenes.begin(); i !=
		scenes.end(); i++){
		delete *i;
	}
	asetettava aliluokkaan, koska aliluokka vastaa Theron::Frameworkin
	hallinnoimisesta
	*/
	~SceneManager();
	/** Hakee globaalin pelialueiden hallintasingletonin Theron-
	osoitteen
	@returns	Pelialueiden hallintaolion osoite
	*/
	static Theron::Address getGlobalManager(){ return globalManager; }
	/** Asettaa aktiivisen pelialueen
	@param scene  pelialueen tunnusnumero
	*/
	virtual bool setActiveScene(unsigned int scene);
	/** Luo uuden pelialueen
	@returns	Uuden pelialueen Theron-osoite
	*/
	virtual Theron::Address createScene();
	/** Hakee viitteen hallintaolion käyttämään Theron::Framework-
	olioon
	*/
	virtual Theron::Framework& getFramework() = 0;
protected:
	/** Pelialuelista*/
	std::vector<Scene*> scenes;
	/** Globaalin pelialueiden hallintasingletonin Theron-osoite*/
	static Theron::Address globalManager;
	/** Aktiivisen pelialueen Theron-osoite*/
	Theron::Address activeScene;
```

```
		/** Olio, joka vastaanottaa hallintaoliolle lähetetyt viestit*/
		Theron::Receiver receiver;
		/** Olio, joka käsittelee hallintaolion vastaanottamat AllDone-
		viestit*/
		Theron::Catcher<AllDone> doneCatcher;
		/** Olio, joka käsittelee hallintaolion vastaanottamat Abort-
		viestit*/
		Theron::Catcher<Abort> abortCatcher;
		/** Olio, joka käsittelee hallintaolion vastaanottamat
		SetActiveScene-viestit*/
		Theron::Catcher<SetActiveScene> setSceneCatcher;
		/** Olio, joka käsittelee hallintaolion vastaanottamat CreateScene-
		viestit*/
		Theron::Catcher<CreateScene> createSceneCatcher;
	};
}
#endif
```

SceneManager.cpp:

```
#include "stdafx.h"
#include "SceneManager.h"
#include <iostream>
#include <LogActor.h>

using namespace stage;

SceneManager::SceneManager(){
	receiver.RegisterHandler(&doneCatcher, &Theron::Catcher<AllDone>::Push);
	receiver.RegisterHandler(&abortCatcher, &Theron::Catcher<Abort>::Push);
	receiver.RegisterHandler(&setSceneCatcher,
	&Theron::Catcher<SetActiveScene>::Push);
	receiver.RegisterHandler(&createSceneCatcher,
	&Theron::Catcher<CreateScene>::Push);
}
SceneManager::~SceneManager(){}
Theron::Address SceneManager::createScene(){
	Scene* scene = new Scene(getFramework());
	scenes.push_back(scene);
	return scene->GetAddress();
}
bool SceneManager::setActiveScene(unsigned int scene){
	if (scene >= scenes.size()){
		getFramework().Send(LogActor::LogError("Error: Attempted to
			activate a scene that does not exist"),
			receiver.GetAddress(), LogActor::getGlobalLogger());
		return false;
	}
	else{
		activeScene = scenes[scene]->GetAddress();
		return true;
	}
}
Theron::Address SceneManager::globalManager = Theron::Address::Null();
```

stdafx.h:

```
#pragma once

#include "targetver.h"
#include <Theron\Theron.h>
```

```
#define WIN32_LEAN_AND_MEAN
```

Transform.h:

```
#pragma once

#ifndef TRANSFORM_H
#define TRANSFORM_H

#define TRANSFORM_ID 1

#include "stdafx.h"
#include "Component.h"
#include <glm\glm.hpp>
#include <glm\gtc\matrix_transform.hpp>
#include <CoreEvents.h>

namespace stage{
	/** Peliolion sijaintia, suuntaa ja kokoa ylläpitävä
	koodrinaattikomponenttiolio
	Ottaa vastaan viestit:
	Update (palauttaa AllDone)
	Render (palauttaa AllDone)
	Transform::GetMatrix (palauttaa Transform::Matrix)
	Transform::SetMatrix (palauttaa AllDone)
	Transform::Translate (palauttaa AllDone)
	*/
	class Transform : public Component {
	public:
		//---Viestit---

		/** Viesti, joka ilmoittaa Transform-olion tilan vastauksena
		GetMatrix-viestiin*/
		struct Matrix : public Event {
			/** 4x4-matriisi, joka kertoo peliolion sijainnin 3D-
			maailmassa*/
			glm::mat4& matrix;
			Matrix(glm::mat4& mt, Theron::Address originator, uint32_t
				msgID) : matrix(mt), Event(originator, msgID){}
			Matrix(uint64_t msgID, glm::mat4& mt) : matrix(mt),
				Event(msgID){}
		};
		/** Viesti, joka ilmoittaa Transform-olion sijainnin vastauksena
		GetMatrix-viestiin*/
		struct Position : public Event {
			/** 4x4-matriisi, joka kertoo peliolion sijainnin 3D-
			maailmassa*/
			glm::vec3 position;
			Position(glm::vec3& pos, Theron::Address originator,
				uint32_t msgID) : position(pos), Event(originator,
				msgID){}
			Position(uint64_t msgID, glm::vec3 pos) : position(pos),
				Event(msgID){}
		};
		/** Viesti, jolla pyydetään Transform-komponenttia lähettämään
		tiedon nykyisestä tilastaan*/
		struct GetMatrix : public Event {
			GetMatrix(Theron::Address originator, uint32_t msgID) :
				Event(originator, msgID){}
			GetMatrix(uint64_t msgID) : Event(msgID){}
		};
```

```
        /** Viesti, jolla pyydetään Transform-komponenttia lähettämään
        nykyisen sijaintinsa*/
        struct GetPosition : public Event {
               GetPosition(Theron::Address originator, uint32_t msgID) :
                      Event(originator, msgID){}
               GetPosition(uint64_t msgID) : Event(msgID){}
        };
        /** Viesti, jolla pyydetään Transform-komponenttia asettamaan
        itselleen uusi tila*/
        struct SetMatrix : public Event {
               /** 4x4-matriisi, joka kertoo peliolion uuden sijainnin 3D-
               maailmassa*/
               glm::mat4& matrix;
               SetMatrix(Theron::Address originator, uint32_t msgID,
                      glm::mat4& mt) : matrix(mt), Event(originator, msgID)
                      {}
               SetMatrix(uint64_t msgID, glm::mat4& mt) : matrix(mt),
                      Event(msgID){}
        };
        /** Viesti, jolla pyydetään Transform-komponenttia siirtämään
               pelioliotaan tiettyyn suuntaan*/
        struct Translate : public Event {
               /** Vektori, joka kertoo miten paljon ja mihin suuntaan
               oliota siirretään*/
               glm::vec3 vector;
               Translate(Theron::Address originator, uint32_t msgID,
                      glm::vec3 vec) : vector(vec), Event(originator,
                      msgID){}
               Translate(uint64_t msgID, glm::vec3 vec) : vector(vec),
                      Event(msgID){}
        };

        //---Metodit---

        /** Luo uuden koordinaattikomponentin. Katso oikea käyttö
        yliluokasta.
        @param fw            Komponenttia ylläpitävä Theron::Framework
        @param owner  Komponentin omistava peliolio
        @param tr            Komponentin alkutilaa kuvaava 4x4-matriisi
        */
        Transform(Theron::Framework& fw, Theron::Address owner, glm::mat4&
               tr = glm::mat4());
private:
        /** Komponentin omistavan peliolion sijaintia 3D-pelimaailmassa
        kuvaava 4x4-matriisi*/
        glm::mat4 transform;

        /** Palauttaa koordinaattikomponentin tunnusluvun*/
        virtual int id(){ return TRANSFORM_ID; }
        /** Hakee tämän komponentin sisäistä tilaa kuvaavan matriisin
        @param msg           Tilan hakupyyntö
        @param sender Pyynnön lähettäjä
        */
        void getMatrix(const GetMatrix& msg, Theron::Address sender);
        /** Hakee tämän komponentin sijaintia pelimaailmassa kuvaavan
        vektorin
        @param msg           Sijainnin hakupyyntö
        @param sender Pyynnön lähettäjä
        */
        void getPosition(const GetPosition& msg, Theron::Address sender);
        /** Asettaa tämän komponentin sisäistä tilaa kuvaavan matriisin
```

```
                @param msg          Tilan muutospyyntö
                @param sender Pyynnön lähettäjä
                */
                void setMatrix(const SetMatrix& msg, Theron::Address sender);
                /** Siirtää tämän komponentin kuvaamia koordinaatteja haluttuun
                suuntaan
                @param msg          Siirtopyyntö
                @param sender Pyynnön lähettäjä
                */
                void translate(const Translate& msg, Theron::Address sender);
        };
}
#endif
```

Transform.cpp:

```
#include "stdafx.h"
#include "Transform.h"

using namespace stage;

Transform::Transform(Theron::Framework& fw, Theron::Address owner, glm::mat4& tr)
        : Component(fw, owner), transform(tr){
        RegisterHandler(this, &Transform::getMatrix);
        RegisterHandler(this, &Transform::setMatrix);
        RegisterHandler(this, &Transform::translate);
        RegisterHandler(this, &Transform::getPosition);
}
void Transform::getMatrix(const GetMatrix& msg, Theron::Address sender){
        Send(Matrix(msg.id, transform), sender);
}
void Transform::getPosition(const GetPosition& msg, Theron::Address sender){
        glm::vec3 position = glm::vec3(transform[3]);
        Send(Position(msg.id, position), sender);
}
void Transform::setMatrix(const SetMatrix& msg, Theron::Address sender){
        transform = msg.matrix;
        Send(AllDone(msg.id), sender);
}
void Transform::translate(const Translate& msg, Theron::Address sender){
        transform = glm::translate(transform, msg.vector);
        Send(AllDone(msg.id), sender);
}
```

Liite 3. Yksisäikeisen vertailumoottorin toteutus

Tämä liite sisältää yksisäikeisen Stage Control -pelimoottoritoteutuksen ohjelmakoodin.

Moduuli Stage_control_core:

CameraComponent.h:

```
#ifndef CAMERACOMPONENT_H
#define CAMERACOMPONENT_H

#define CAMERA_ID 2

#include "stdafx.h"
#include <Component.h>
#include <Camera.h>
#include <Transform.h>
#include <glm\gtc\matrix_transform.hpp>

namespace stage_control{
	/** Peliolioon liitettävä komponenttiolio, joka mallintaa kameran.*/
	class CameraComponent : public Component {
	public:
		/** Luo uuden kamerakomponentin oletusasetuksilla (45 asteen fov,
		kuvasuhde 4:3, piirtoetäisyys 0.1-100, suunnattu origoon).
		Katso oikea käyttö yliluokasta.
		@param owner  Osoitin tämän komponentin omistavaan peliolioon
		*/
		CameraComponent(GameObject* owner);
		/** Luo uuden kamerakomponentin ja asettaa kameran alkutilan. Katso
		oukea käyttö yliluokasta.
		@param owner               Osoitin tämän komponentin
		                           omistavaan peliolioon
		@param initialProjection   Kameran projektiomatriisi simulaation
		                           alussa
		@param initialView         Kameran näkymämatriisi simulaation
		                           alussa
		*/
		CameraComponent(GameObject* owner, glm::mat4& initialProjection,
			glm::mat4& initialView);
		/** Palauttaa kamerakomponentin komponenttitunnuksen
		@returns      Kamerakomponentin komponenttitunnus
		*/
		virtual int id(){ return CAMERA_ID; }
		/** Valmistelee kameran ruudun piirtämistä varten*/
		virtual void render();
		/** Hakee osoittimen kamerakomponentin kameraolioon
		@returns      osoitin kameraolioon
		*/
		stage_common::Camera* getRawCamera(){ return &cam; }
	private:
		/** Osoitin kamerakomponentin omistavan peliolion sijaintia
		ylläpitävään olioon  */
		Transform* position;
		/** Kameran tilaa ylläpitävä olio*/
		stage_common::Camera cam;
	};
```

```
}
#endif
```

CameraComponent.cpp:

```
#include "stdafx.h"
#include "CameraComponent.h"
#include <SceneManager.h>

using namespace stage_control;

CameraComponent::CameraComponent(GameObject* owner) : Component(owner){
	position = (Transform*)owner->getComponentByID(TRANSFORM_ID);
	if (position == nullptr){
		SceneManager::getGlobalLogger().LogError("Error: parent game object
			does not have a transform, can't initialize camera
			component");
		abort();
	}
}
CameraComponent::CameraComponent(GameObject* owner, glm::mat4& initialProjection,
	glm::mat4& initialView) : Component(owner), cam(initialProjection,
	initialView){
	position = (Transform*)owner->getComponentByID(TRANSFORM_ID);
	if (position == nullptr){
		SceneManager::getGlobalLogger().LogError("Error: parent game object
			does not have a transform, can't initialize camera
			component");
		abort();
	}
}
void CameraComponent::render(){
	cam.setViewMatrix(position->getMatrix());
}
```

Gameloop.h:

```
#ifndef GAMELOOP_H
#define GAMELOOP_H

#include "stdafx.h"
#include <Scene.h>
#include <SceneManager.h>
#include <GraphicsController.h>
#include <Timer.h>
#include "CameraComponent.h"

namespace stage_control{
	/** Pelisilmukan mallintava luokka
	*/
	class Gameloop : public SceneManager{
	public:
		/** Luo uuden pelisilmukan ja avaa ikkunan
		@param windowName	Ikkunan nimi
		@param xres			Ikkunan vaakaresoluutio
		@param yres			Ikkunan pystyresoluutio
		*/
		Gameloop(std::string& windowName, int xres, int yres);
		/** Hakee pelisilmukan aikaskaalan (pelin simulointinopeus)
		@returns	Pelin aikaskaala
		*/
		float getTimescale();
```

```
		/** Asettaa pelisilmukan aikaskaalan(pelin simulointinopeus)
		@returns	Pelin aikaskaala
		*/
		void setTimescale(float ts);
		/** Käynnistää pelisilmukan suorituksen*/
		void start();
		/** Pysäyttää pelisilmukan suorituksen. Kutsuttava ennen
		pelisilmukan tuhoamista.*/
		void stop();
		/** Asettaa pelin aktiivisen pelialueen
		@param scene  Aktiivinen pelialue
		*/
		void setActiveScene(Scene* scene);
		/** Asettaa pelin aktiivisen kameran
		@param scene  Aktiivinen kamera
		*/
		void setActiveCamera(CameraComponent* cam);
		/** Palauttaa pelimoottorin käynnistämisestä kuluneiden
		ruudunpäivitysten määrän
		@returns	Pelimoottorin käynnistämisestä kuluneiden
				ruudunpäivitysten määrä
		*/
		unsigned int getCurrentFrame();
		/** Tuhoaa pelisilmukkaolion. Kutsu ensin stop-metodia, jotta
		pelisilmukka voidaan tuhota turvallisesti.*/
		~Gameloop();
	private:
		/** Aktiivinen pelialue, eli se pelialue, jonka tilaa päivitetään
		ja joka piirretään ruudulle*/
		Scene* activeScene;
		/** Pelin grafiikkamoottoria hallinnoiva olio*/
		stage_common::GraphicsController* gc;
		/** Pelin aktiivinen kamera, eli se kamera, jonka kuvakulmasta
		pelimaailma näytetään*/
		stage_common::Camera* cam;
		/** Pelisilmukan suoritusaikaa mittaava ajastin*/
		stage_common::Timer loopTimer;
		/** Lukitaanko hiiri keskelle kuvaruutua?*/
		bool resetMouse = false;
		/** Pelin aikaskaala, eli pelisimulaation suoritusnopeus*/
		float timescale = 1;
		/** Ilmoittaa, pitääko pelisilmukan suoritus pysäyttää nykyisen
		silmukan jälkeen*/
		bool abort = false;
		/** Suorittaa pelisilmukkaa, kunnes pysäytysmetodia kutsutaan*/
		void loop();
		/** Viimeistelee pelisilmukan suorituksen lopettamisen ja
		varmistaa, että pelisilmukka voidaan tuhota turvallisesti*/
		void shutdown();
	};
}
#endif
```

Gameloop.cpp:

```
#pragma once

#include "stdafx.h"
#include <iostream>
#include <string>
#include "Gameloop.h"
```

```
#include <Timer.h>
#include <Input.h>

using namespace stage_control;

Gameloop::Gameloop(std::string& windowName, int xres, int yres) :
	activeScene(nullptr){
	gc = new stage_common::GraphicsController(windowName, xres, yres);
	if (globalLogger != nullptr){
		globalLogger->LogError("Error: global logger already set.
			Aborting.");
		std::abort();
	}
	globalLogger = new stage_common::Logger(std::cout, std::cerr);
	if (SceneManager::getGlobalManager() != nullptr){
		globalLogger->LogError("Error: global scene manager already set.
			Aborting.");
		std::abort();
	}
	SceneManager::setGlobalManager(this);
}
Gameloop::~Gameloop(){
	if (SceneManager::getGlobalManager() == this)
		SceneManager::setGlobalManager(NULL);
	delete gc;
	delete globalLogger;
}
float Gameloop::getTimescale() {
	return timescale;
}
void Gameloop::setTimescale(float ts) {
	timescale = ts;
}
void Gameloop::start() {
	if (activeScene != nullptr && cam !=nullptr)
		loop();
	else{
		std::cout << "Active scene or camera not defined, shutting down" <<
			std::endl;
		std::abort();
	}
}
void Gameloop::stop(){
	abort = true;
}
void Gameloop::loop() {
	stage_common::Timer upTimer;
	stage_common::Timer rendTimer;
	stage_common::Timer maintTimer;
	while (!abort) {
		loopTimer.start();
		//Päivitysvaihe
		upTimer.start();
		activeScene->update(loopTimer.lastTickTime());
		upTimer.stop();
		//Piirtovaihe
		rendTimer.start();
		activeScene->render();
		gc->draw(*cam);
		rendTimer.stop();
		//Ylläpitovaihe
```

```
			maintTimer.start();
			if (gc->shouldStop()) abort = true;
			stage_common::Input::getSingleton().update(resetMouse);
			maintTimer.stop();
			loopTimer.stop();
		}
		std::cout << "Total runtime: " << loopTimer.totalTime() << std::endl;
		std::cout << "Total frames: " << loopTimer.totalTicks() << std::endl;
		std::cout << "Average loop time: " << loopTimer.averageTime() <<
			std::endl;
		std::cout << "Average fps: " << std::to_string(1000 /
			loopTimer.averageTime()) << std::endl;
		std::cout << "Average update time: " << upTimer.averageTime() <<
			std::endl;
		std::cout << "Average render time: " << rendTimer.averageTime() <<
			std::endl;
		std::cout << "Average maintenance time: " << maintTimer.averageTime() <<
			std::endl;
		shutdown();
}
void Gameloop::shutdown(){
		std::cout << "shutting down";
}
void Gameloop::setActiveScene(Scene* scene){
		activeScene = scene;
}
void Gameloop::setActiveCamera(CameraComponent* cam){
		this->cam = (cam->getRawCamera());
}
unsigned int Gameloop::getCurrentFrame(){
		return loopTimer.totalTicks();
}
```

ModelComponent.h:

```
#ifndef MODELCOMPONENT_H
#define MODELCOMPONENT_H

#define MODEL_ID 3

#include "stdafx.h"
#include <Component.h>
#include <GameObject.h>
#include <Model.h>
#include <Transform.h>
#include <GraphicsController.h>
#include <Shader.h>

namespace stage_control{
		/** Komponentti, joka mahdollistaa 3D-mallin liittämisen peliolioon*/
		class ModelComponent : public Component {
		public:
			/** Luo uuden mallikomponentin. Katso oikea käyttö yliluokasta.
			@param owner  Osoitin tämän olion omistavaan peliolioon
			@param mod    Osoitin siihen 3D-malliin, joka halutaan
			              liittää tämän olion omistajaan
			*/
			ModelComponent(GameObject* owner, stage_common::Model* mod);
			/** Tuhoaa mallikomponentin */
			~ModelComponent(){}
			/** Valmistelee mallin piirrettäväksi ruudulle*/
```

```
		virtual void render();
		/** Palauttaa mallikomponentin komponenttitunnuksen
		@returns	Mallikomponentin komponenttitunnus
		*/
		virtual int id(){ return MODEL_ID; }
	private:
		/** Osoitin tämän mallikomponentin 3D-malliin (Jaettu kaikkien
		samaa mallia käyttävien ModelComponent:ien kesken)*/
		stage_common::Model* mod;
		/** Osoitin olioon, joka esittää tämän komponentin omistavan
		peliolion sijaintia 3D-maailmassa*/
		Transform* position;
	};
}
#endif
```

ModelComponent.cpp:

```
#include "stdafx.h"
#include "ModelComponent.h"

using namespace stage_control;

ModelComponent::ModelComponent(GameObject* owner, stage_common::Model* mod) :
	Component(owner), mod(mod){
	position = (Transform*)owner->getComponentByID(TRANSFORM_ID);
	if (position == nullptr) abort();
}
void ModelComponent::render(){
	stage_common::GraphicsController::getGlobalController()->queue(mod,
		position->getMatrix());
}
```

PhysicsComponent.h:

```
#ifndef PHYSICSCOMPONENT_H
#define PHYSICSCOMPONENT_H

#include "stdafx.h"
#include <Component.h>
#include <Collider.h>
#include <SphereCollider.h>
#include <AABBCollider.h>
#include <Transform.h>
#include <EventChannel.h>

#define PHYSICS_COLLISION_EVENT_TYPE 99001
#define PHYSICSCOMPONENT_ID 4

namespace stage_control{
	/** Peliolioon liitettävä komponentti, joka liittää peliolioon nopeuden ja
	törmäystentunnistuksen*/
	class PhysicsComponent : public Component, public EventHandler{
	public:
		/** Viesti, jolla kysytään, onko lähettäjä törmännyt
		vastaanottajaan*/
		struct CollisionEvent : public Event{
			virtual unsigned int getEventType() const {
				return PHYSICS_COLLISION_EVENT_TYPE;
			};
			/** Viite lähettäjän törmäyshahmoon*/
			const stage_common::Collider& collider;
```

```
        /** Viite lähettäjäolioon*/
        PhysicsComponent& sender;
        /** Edellisestä ruudunpäivityksestä kulunut aika*/
        float elapsedMS;
        CollisionEvent(stage_common::Collider& coll,
                PhysicsComponent& sender, float elapsedMS)
                : collider(coll), sender(sender),
                elapsedMS(elapsedMS){}
};
/** Luo uuden fysiikkakomponentin, joka käyttää pallotörmäyshahmoa
@param owner  Komponentin omistava peliolio
@param radius Törmäyshahmon säde
@param initialV      Peliolion nopeus simulaation alussa
@param mass          Peliolion massa
*/
PhysicsComponent(GameObject* owner, float radius, glm::vec3
        initialV, float mass):
        Component(owner), velocity(initialV), mass(mass){
        setup(owner);
        collider = new stage_common::SphereCollider(radius,
        transform->getPosition());
}
/** Luo uuden fysiikkakomponentin, joka käyttää AABB-törmäyshahmoa
@param owner  Komponentin omistava peliolio
@param radius Törmäyshahmon säde
@param initialV      Peliolion nopeus simulaation alussa
@param mass          Peliolion massa
*/
PhysicsComponent(GameObject* owner, glm::vec3 size, glm::vec3
        initialV, float mass) :
        Component(owner), velocity(initialV), mass(mass){
        setup(owner);
        collider = new stage_common::AABBCollider(size, transform-
        >getPosition());
}
/** Päivittää fysiikkakomponentin tilan
@param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
                     millisekunteina
*/
void update(float elapsedMS){
        //Siirretään törmäyshahmoa vain, jos tilaa ei ole vielä
        //päivitetty törmäysviestin seurauksena
        if (!updatedThisFrame){
                updatePosition(elapsedMS);
        }
        //Lähetetään muille fysiikkakomponenteille tieto
        //törmäyshahmon liikkeestä
        CollisionEvent ev(*collider, *this, elapsedMS);
        collisionChannel().broadcastOthers(ev, this);
        //Päivitetään peliolion sijainti vastaamaan törmäyshahmon
        //uutta sijaintia
        transform->translate(collider->center - oldPos);
}
void render(){
        //Resetoidaan tilan päivittämisen kieltävä muuttuja update-
        //kutsujen välissä
        updatedThisFrame = false;
}
/** Käsittelee olion vastaanottamat tapahtumaviestit
@param ev     Käsiteltävä viesti
*/
```

```
        void handleEvent(const Event& ev){
                //Käsitellään vain törmäysviestit
                if (ev.getEventType() != PHYSICS_COLLISION_EVENT_TYPE)
                        return;
                const CollisionEvent& coll = (const CollisionEvent&)ev;
                //Jos tilaa ei vielä ole päivitetty tämän ruudunpäivityksen
                //aikana, päivitetään se nyt
                if (!updatedThisFrame){
                        updatePosition(coll.elapsedMS);
                }
                //Tarkistetaan, onko törmäys tapahtunut
                if (!collider->checkCollision(coll.collider)) return;
                //Jos törmäys on tapahtunut, lasketaan molemmille
                //kappaleille uudet nopeudet
                glm::vec3 otherNewV = coll.sender.getVelocity();
                stage_common::Collisions::collisionVelocityChange(velocity,
                        mass, otherNewV, coll.sender.getMass());
                //Pyydetään toista kappaletta päivittämään tilansa
                coll.sender.processCollision(*collider, otherNewV);
        }
        /** Päivittää peliolion tilan ottamaan huomioon tapahtunut törmäys
        @param coll   Se törmäyshahmo, johon tämä olio törmäsi
        @param newV   Tämän olion uusi nopeus törmäyksen jälkeen
        */
        void processCollision(const stage_common::Collider& coll, glm::vec3
                newV){
                //Siirretään oliota, kunnes se ei enää törmää
                stage_common::Collisions::backOff(*collider, -1.0f * newV,
                        coll);
                //Asetetaan uusi nopeus
                velocity = newV;
        }
        /** Hakee tämän olion nykyisen nopeuden
        @returns      Olion nopeusvektori
        */
        glm::vec3 getVelocity(){ return velocity; }
        /** Hakee tämän olion massan
        @returns      Olion massa
        */
        float getMass(){ return mass; }
        /**Tuhoaa fysiikkakomponentin*/
        ~PhysicsComponent(){
                delete collider;
        }
        /** Palauttaa fysiikkakomponentin komponenttitunnuksen
        @returns      Komponentin komponenttitunnus
        */
        int id(){
                return PHYSICSCOMPONENT_ID;
        }
        /** Hakee viitteen fysiikkamoottorin törmäystestien
        tapahtumakanavaan
        @returns      Viite tapahtumakanavaan
        */
        static EventChannel& collisionChannel(){
                //Staattinen kanava-singleton, jaettu kaikkien
                //fysiikkakomponenttien kesken
                static EventChannel cc;
                return cc;
        }
private:
```

```
		/** Tämän peliolion törmäyshahmo*/
		stage_common::Collider* collider;
		/** Osoitin tämän komponentin omistajaolion pelimaailmasijaintia
		ylläpitävään komponenttiin*/
		Transform* transform;
		/** Tämän fysiikkaolion liikesuunta ja nopeus*/
		glm::vec3 velocity;
		/** Tämän fysiikkaolion liikesuunta ja nopeus nykyisen
		ruudunpäivityksen alussa*/
		glm::vec3 oldPos;
		/** Tämän fysiikkaolion massa*/
		float mass;
		/** Ilmoittaa, onko tämän peliolion tila jo päivitetty tämän
		ruudunpäivityksen aikana*/
		bool updatedThisFrame = false;
		/** Molemmille konstruktoreille yhteinen apumetodi, joka alustaa
		peliolion tilan
		@param owner  Tämän komponentin omistava peliolio
		*/
		void setup(GameObject* owner){
			transform = (Transform*)(owner
				->getComponentByID(TRANSFORM_ID));
			collisionChannel().registerRecipient(this);
		}
		/** Päivittää fysiikkaolion sijainnin pelimaailmassa olion nopeuden
		perusteella
		@param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
		*/
		void updatePosition(float elapsedMS){
			oldPos = transform->getPosition();
			collider->center = oldPos + (velocity * elapsedMS);
			updatedThisFrame = true;
		}
	};
}
#endif
```

StaticGeometryComponent.h:

```
#pragma once

#ifndef STATICGEOMETRYCOMPONENT_H
#define STATICGEOMETRYCOMPONENT_H

#define STATICGEOMETRYCOMPONENT_ID 5

#include "PhysicsComponent.h"

namespace stage_control{
	/** Pelioliokomponentti, joka mallintaa staattista, liikkumatonta
	törmäyshahmoa, kuten pelimaailman maastoa
	tai seiniä.*/
	class StaticGeometryComponent : public Component, public EventHandler{
	public:
		/** Luo uuden staattisen törmäyshahmon, joka käyttää
		pallotörmäyshahmoa
		@param owner  Komponentin omistava peliolio
		@param radius Törmäyshahmon säde
		*/
		StaticGeometryComponent(GameObject* owner, float radius) :
			Component(owner){
```

```
                setup(owner);
                collider = new stage_common::SphereCollider(radius,
                        transform->getPosition());
        }
        /** Luo uuden staattisen törmäyshahmon, joka käyttää AABB-
        törmäyshahmoa
        @param owner  Komponentin omistava peliolio
        @param radius Törmäyshahmon koko
        */
        StaticGeometryComponent(GameObject* owner, glm::vec3 size) :
                Component(owner){
                setup(owner);
                collider = new stage_common::AABBCollider(size, transform
                        ->getPosition());
        }
        /** Päivittää komponentin tilan
        @param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
        */
        void update(float elapsedMS){
                //Haetaan uudelleen sijainti, sillä jokin muu komponentti on
                //voinut muuttaa sitä
                collider->center = transform->getPosition();
        }
        /** Käsittelee olion vastaanottamat tapahtumaviestit
        @param ev     Käsiteltävä viesti
        */
        void handleEvent(const Event& ev){
                //Käsitellään vain törmäysviestit
                if (ev.getEventType() != PHYSICS_COLLISION_EVENT_TYPE)
                        return;
                const PhysicsComponent::CollisionEvent& coll = (const
                        PhysicsComponent::CollisionEvent&)ev;
                //Tarkistetaan, onko törmäys tapahtunut
                if (!collider->checkCollision(coll.collider)) return;
                glm::vec3 otherV = coll.sender.getVelocity();
                //Kimmotetaan toinen kappale tästä
                otherV = stage_common::Collisions::reflect(otherV, collider
                        ->getCollisionNormal(coll.collider, otherV));
                //Pyydetään toista kappaletta päivittämään tilansa
                coll.sender.processCollision(*collider, otherV);
        }
        /** Palauttaa staattisen törmäyshahmokomponentin
        komponenttitunnuksen
        @returns      Komponentin komponenttitunnus
        */
        int id(){
                return STATICGEOMETRYCOMPONENT_ID;
        }
        /** Tuhoaa staattisen törmäyshahmokomponentin*/
        ~StaticGeometryComponent(){
                delete collider;
        }
private:
        /** Komponentin törmäyshahmo
        */
        stage_common::Collider* collider;
        /** Osoitin tämän komponentin omistajaolion pelimaailmasijaintia
        ylläpitävään komponenttiin
        */
        Transform* transform;
        /** Kaikille konstruktoreille yhteinen apumetodi, joka alustaa
```

```
		peliolion tilan
		@param owner  Tämän komponentin omistava peliolio
		*/
		void setup(GameObject* owner){
			transform = (Transform*)(owner
				->getComponentByID(TRANSFORM_ID));
			PhysicsComponent::collisionChannel()
				.registerRecipient(this);
		}
	};
}
#endif
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN
```

Moduuli Stage_control_demo:

CameraControlComponent.h:

```
#pragma once

#include "stdafx.h"
#ifndef CAMERACONTROLCOMPONENT_H
#define CAMERACONTROLCOMPONENT_H

#define CAMERACONTROLCOMPONENT_ID 6
/** Kameran liikenopeus*/
#define CAMERASPEED 0.025f

#include <Component.h>
#include <Transform.h>
#include <Input.h>
#include <GLFW\glfw3.h>
#include <SceneManager.h>
#include <glm\gtx\quaternion.hpp>

namespace stage_control{
	/** Peliolioon liitettävä komponentti, joka mahdollistaa peliolion
	liikuttamisen näppäimistöllä.
	Suunniteltu kameran liikuttamista varten.*/
	class CameraControlComponent : public Component{
	public:
		/** Luo uuden ohjauskomponentin
		@param owner  Komponentin omistava olio
		*/
		CameraControlComponent(GameObject* owner) : Component(owner){
			tr = (Transform*)(owner->getComponentByID(TRANSFORM_ID));
			//Rekisteröidään tarkkailtavat näppäimet
			stage_common::Input& in =
				stage_common::Input::getSingleton();
			in.registerKey(GLFW_KEY_W);
			in.registerKey(GLFW_KEY_S);
			in.registerKey(GLFW_KEY_A);
			in.registerKey(GLFW_KEY_D);
			in.registerKey(GLFW_KEY_R);
```

```
                in.registerKey(GLFW_KEY_F);
                in.registerKey(GLFW_KEY_ESCAPE);
        }
        /** Päivittää komponentin tilan
        @param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
        */
        void update(float elapsedMS){
                stage_common::Input& in =
                        stage_common::Input::getSingleton();
                //Suunta, johon liikutaan
                glm::vec3 movement;
                //Tarkistetaan mitä näppäimiä on painettu
                if (in.getKeyDown(GLFW_KEY_W)) movement.z += CAMERASPEED *
                        elapsedMS;
                if (in.getKeyDown(GLFW_KEY_S)) movement.z -= CAMERASPEED  *
                        elapsedMS;
                if (in.getKeyDown(GLFW_KEY_A)) movement.x += CAMERASPEED  *
                        elapsedMS;
                if (in.getKeyDown(GLFW_KEY_D)) movement.x -= CAMERASPEED  *
                        elapsedMS;
                if (in.getKeyDown(GLFW_KEY_F)) movement.y += CAMERASPEED  *
                        elapsedMS;
                if (in.getKeyDown(GLFW_KEY_R)) movement.y -= CAMERASPEED  *
                        elapsedMS;
                //Siirretään isäntäoliota
                tr->translate(movement);
                //Suljetaan peli, jos painetaan escapea
                if (in.getKeyDown(GLFW_KEY_ESCAPE))
                        SceneManager::getGlobalManager()->stop();
        }
        /** Hakee tämän komponentin komponenttitunnuksen
        @returns      Komponentin tunnus
        */
        int id(){
                return CAMERACONTROLCOMPONENT_ID;
        }
    private:
        /** Isäntäolion sijaintia hallinnoiva olio*/
        Transform* tr;
    };
}
#endif
```

GameObjectFactory.h:

```
#ifndef GAMEOBJECTFACTORY_H
#define GAMEOBJECTFACTORY_H

#include "stdafx.h"
#include "Plane.h"
#include "Sphere.h"
#include <Scene.h>
#include <glm\glm.hpp>
#include <PhysicsComponent.h>
#include <StaticGeometryComponent.h>
#include <ModelComponent.h>
#include <SimpleShader.h>
#include "Waiter.h"

namespace stage_control{
    /** Luokka, jota käytetään demo-ohjelman peliolioiden luomiseen*/
```

```
class GameObjectFactory{
public:
	/** Luo satunnaiseen paikkaan pallon, joka liikkuu satunnaiseen
	suuntaan
	@param sc                          Pelialue, jolle pallo luodaan
	@param maxCoordinates      Luotavan pallon maksimietäisyys origosta
	@returns                           Viite luotuun peliolioon
	*/
	static GameObject& constructRandomSphere(Scene* sc, glm::vec3
		maxCoordinates, int waitLimit){
		//Luodaan peliolio
		GameObject& obj = sc->createObject();
		//Arvotaan peliolion alkusijainti
		glm::vec3 translation(randomFloat(-maxCoordinates.x,
			maxCoordinates.x),
			randomFloat(-maxCoordinates.y, maxCoordinates.y),
			randomFloat(-maxCoordinates.z, maxCoordinates.z));
		glm::mat4 transform = glm::translate(glm::mat4(1.0f),
			translation);
		//Luodaan peliolille sijaintikomponentti
		Transform* tf = new Transform(&obj, transform);
		//Kiinnitetään peliolioon 3D-malli
		ModelComponent* mod = new ModelComponent(&obj,
			&(getSingleton().mod_sphere));
		//Arvotaan peliolion alkunopeus
		glm::vec3 velocity(randomFloat(-0.01f, 0.01f), randomFloat(-
			0.01f, 0.01f), randomFloat(-0.01f, 0.01f));
		//Luodaan peliolille fysiikkakomponentti
		PhysicsComponent* pc = new PhysicsComponent(&obj, 1.0f,
			velocity, 1.0f);
		//Luodaan tarvittaessa Waiter-komponentti
		if (waitLimit > 0) Waiter* w = new Waiter(&obj, waitLimit);
		return obj;
	}
	/** Luo pelimaailmaan seinän eli litteän, staattisen törmäyspinnan
	@param sc                  Pelialue, johon seinä luodaan
	@param transform     Seinän keskipisteen 3D-sijainti
	@param size                Seinän koko
	@returns                           Viite luotuun peliolioon
	*/
	static GameObject& constructWall(Scene* sc, const glm::mat4&
		transform, glm::vec3 size){
		//Luodaan uusi peliolio
		GameObject& obj = sc->createObject();
		//Luodaan peliolille sijaintikomponentti
		Transform* tf = new Transform(&obj, transform);
		//Kiinnitetään peliolioon 3D-malli
		ModelComponent* mod = new ModelComponent(&obj,
		&(getSingleton().mod_plane));
		//Luodaan peliolille liikkumaton törmäyshahmo
		StaticGeometryComponent* sgc = new
		StaticGeometryComponent(&obj, size);
		return obj;
	}
private:
	/** Demo-ohjelman peliolioiden käyttämä sävytinohjelma*/
	stage_common::SimpleShader ss;
	/** Pallon 3D-malli  */
	stage_common::Model mod_sphere;
	/** Seinän 3D-malli  */
	stage_common::Model mod_plane;
```

```
		/**Hakee viitteen globaaliin singleton-olioon, joka luo 3D-mallit
		ja niiden sävytinohjelman sekä pitää ne muistissa
		@returns	Viite singleton-olioon
		*/
		static GameObjectFactory& getSingleton(){
			//Luodaan singleton ensimmäisellä suorituskerralla
			static GameObjectFactory gof;
			return gof;
		}
		/**Arpoo liukuluvun kahden liukuluvun väliltä
		@param start  Arvottavan luvun alaraja
		@param end		Arvottavan luvun yläraja
		@returns		Satunnainen luku ala- ja ylärajojen väliltä
		*/
		static float randomFloat(float start, float end){
			float random = ((float)rand()) / (float)RAND_MAX;
			return start + (end - start) * random;
		}
		/** Luo uuden GameObjectFactory-olion, eli käytännössä lataa
		muistiin peliolioiden luomisessa tarvittavat resurssit */
		GameObjectFactory() : mod_sphere(generate_sphere_vertices(),
			generate_sphere_colors(), &ss),
			mod_plane(generate_plane_vertices(),
			generate_plane_colors(), &ss){
		}
	};
}
#endif
```

Plane.h:

```
#ifndef PLANE_H
#define PLANE_H

#include "stdafx.h"
#include <vector>
#include <glm\glm.hpp>

/*Sisältää litteän pinnan 3D-mallin piirtämiseen tarvittavan datan*/
namespace stage_control{
	/** Litteän pinnan verteksit*/
	static std::vector<glm::vec3> plane_vertices = {
		glm::vec3(-1.000000, 0.000000, 1.000000),
		glm::vec3(1.000000, 0.000000, 1.000000),
		glm::vec3(-1.000000, 0.000000, - 1.000000),
		glm::vec3(1.000000, 0.000000, - 1.000000)
	};
	/** Pinnan tahkot
	Jokainen kolmen numeron sarja määrittelee yhden kolmion, jonka kärjet ovat
	numeroiden ilmaisemat verteksit.*/
	static std::vector<int> plane_faces = {
		2, 4, 3,
		1, 2, 3
	};
	/** Yhdistää tahkot ja verteksit yhtenäiseksi verteksilistaksi
	@returns	Litteää pintaa kuvaavan 3D-mallin verteksit
	*/
	static std::vector<glm::vec3> generate_plane_vertices(){
		std::vector<glm::vec3> ret;
		//Jokaista tahkoa kohden: lisää listaan järjestyksessä tahkon
		//verteksit
```

```
		for (unsigned int i = 0; i < plane_faces.size(); i++){
			ret.push_back(plane_vertices[plane_faces[i] - 1]);
		}
		return ret;
	}
	/** Arpoo vertekseille väriarvot
	@returns	Litteää pintaa kuvaavan 3D-mallin värit
	*/
	static std::vector<glm::vec3> generate_plane_colors(){
		std::vector<glm::vec3> ret;
		for (unsigned int i = 0; i < plane_faces.size(); i++){
			ret.push_back(glm::vec3(static_cast <float> (rand()) /
				static_cast <float> (RAND_MAX),
				static_cast <float> (rand()) / static_cast <float>
				(RAND_MAX),
				static_cast <float> (rand()) / static_cast <float>
				(RAND_MAX)));
		}
		return ret;
	}
}
#endif
```

simplefragmentshader.glsl:

```
//Pikselisävytin
#version 330 core

//Parametri: värit
in vec3 fragmentColor;
//Palautetaan lasketut värit
out vec3 color;

void main(){
    color = fragmentColor;
}
```

simplevertexshader.glsl:

```
//Verteksisävytin
#version 330 core

//1. parametri: verteksit
layout(location = 0) in vec3 vertexPosition_modelspace;
//2. parametri: värit
layout(location = 1) in vec3 vertexColor;

//Annetaan värit pikselisävyttimelle
out vec3 fragmentColor;

//Mallin sijainti suhteessa kameraan
uniform mat4 MVP;

//Lasketaan verteksin sijainti ruudulla suhteessa malliin ja annetaan värit
//pikselisävyttimelle
void main(){
    vec4 v = vec4(vertexPosition_modelspace,1);
    gl_Position = MVP * v;
       fragmentColor = vertexColor;
}
```

Sphere.h:

```
#ifndef SPHERE_H
#define SPHERE_H

#include "stdafx.h"
#include <vector>
#include <glm\glm.hpp>

/*Sisältää palloa approksimoivan 3D-mallin piirtämiseen tarvittavan datan*/
namespace stage_control{
	/** Pallon verteksit*/
	static std::vector<glm::vec3> sphere_vertices = {
		glm::vec3(0.000000, -1.000000, 0.000000),
		glm::vec3(0.723607, -0.447220, 0.525725),
		glm::vec3(-0.276388, -0.447220, 0.850649),
		glm::vec3(-0.894426, -0.447216, 0.000000),
		glm::vec3(-0.276388, -0.447220, - 0.850649),
		glm::vec3(0.723607, -0.447220, -0.525725),
		glm::vec3(0.276388, 0.447220, 0.850649),
		glm::vec3(-0.723607, 0.447220, 0.525725),
		glm::vec3(-0.723607, 0.447220, -0.525725),
		glm::vec3(0.276388, 0.447220, -0.850649),
		glm::vec3(0.894426, 0.447216, 0.000000),
		glm::vec3(0.000000, 1.000000, 0.000000),
		glm::vec3(-0.162456, -0.850654, 0.499995),
		glm::vec3(0.425323, -0.850654, 0.309011),
		glm::vec3(0.262869, -0.525738, 0.809012),
		glm::vec3(0.850648, -0.525736, 0.000000),
		glm::vec3(0.425323, -0.850654, -0.309011),
		glm::vec3(-0.525730, -0.850652, 0.000000),
		glm::vec3(-0.688189, -0.525736, 0.499997),
		glm::vec3(-0.162456, -0.850654, -0.499995),
		glm::vec3(-0.688189, -0.525736, -0.499997),
		glm::vec3(0.262869, -0.525738, -0.809012),
		glm::vec3(0.951058, 0.000000, 0.309013),
		glm::vec3(0.951058, 0.000000, -0.309013),
		glm::vec3(0.000000, 0.000000, 1.000000),
		glm::vec3(0.587786, 0.000000, 0.809017),
		glm::vec3(-0.951058, 0.000000, 0.309013),
		glm::vec3(-0.587786, 0.000000, 0.809017),
		glm::vec3(-0.587786, 0.000000, -0.809017),
		glm::vec3(-0.951058, 0.000000, -0.309013),
		glm::vec3(0.587786, 0.000000, -0.809017),
		glm::vec3(0.000000, 0.000000, -1.000000),
		glm::vec3(0.688189, 0.525736, 0.499997),
		glm::vec3(-0.262869, 0.525738, 0.809012),
		glm::vec3(-0.850648, 0.525736, 0.000000),
		glm::vec3(-0.262869, 0.525738, -0.809012),
		glm::vec3(0.688189, 0.525736, -0.499997),
		glm::vec3(0.162456, 0.850654, 0.499995),
		glm::vec3(0.525730, 0.850652, 0.000000),
		glm::vec3(-0.425323, 0.850654, 0.309011),
		glm::vec3(-0.425323, 0.850654, -0.309011),
		glm::vec3(0.162456, 0.850654, -0.499995)
	};
	/** Pallon tahkot
	Jokainen kolmen numeron sarja määrittelee yhden kolmion, jonka kärjet ovat
	numeroiden ilmaisemat verteksit.*/
	static std::vector<int> sphere_faces = {
		1, 14, 13,
		2, 14, 16,
		1, 13, 18,
```

```
1, 18, 20,
1, 20, 17,
2, 16, 23,
3, 15, 25,
4, 19, 27,
5, 21, 29,
6, 22, 31,
2, 23, 26,
3, 25, 28,
4, 27, 30,
5, 29, 32,
6, 31, 24,
7, 33, 38,
8, 34, 40,
9, 35, 41,
10, 36, 42,
11, 37, 39,
39, 42, 12,
39, 37, 42,
37, 10, 42,
42, 41, 12,
42, 36, 41,
36, 9, 41,
41, 40, 12,
41, 35, 40,
35, 8, 40,
40, 38, 12,
40, 34, 38,
34, 7, 38,
38, 39, 12,
38, 33, 39,
33, 11, 39,
24, 37, 11,
24, 31, 37,
31, 10, 37,
32, 36, 10,
32, 29, 36,
29, 9, 36,
30, 35, 9,
30, 27, 35,
27, 8, 35,
28, 34, 8,
28, 25, 34,
25, 7, 34,
26, 33, 7,
26, 23, 33,
23, 11, 33,
31, 32, 10,
31, 22, 32,
22, 5, 32,
29, 30, 9,
29, 21, 30,
21, 4, 30,
27, 28, 8,
27, 19, 28,
19, 3, 28,
25, 26, 7,
25, 15, 26,
15, 2, 26,
23, 24, 11,
23, 16, 24,
```

```
                16, 6, 24,
                17, 22, 6,
                17, 20, 22,
                20, 5, 22,
                20, 21, 5,
                20, 18, 21,
                18, 4, 21,
                18, 19, 4,
                18, 13, 19,
                13, 3, 19,
                16, 17, 6,
                16, 14, 17,
                14, 1, 17,
                13, 15, 3,
                13, 14, 15,
                14, 2, 15
        };
        /** Yhdistää tahkot ja verteksit yhtenäiseksi verteksilistaksi
        @returns      Palloa kuvaavan 3D-mallin verteksit
        */
        static std::vector<glm::vec3> generate_sphere_vertices(){
                std::vector<glm::vec3> ret;
                for (unsigned int i = 0; i < sphere_faces.size(); i++){
                        ret.push_back(sphere_vertices[sphere_faces[i] - 1]);
                }
                return ret;
        }
        /** Arpoo vertekseille väriarvot
        @returns      Litteää pintaa kuvaavan 3D-mallin värit
        */
        static std::vector<glm::vec3> generate_sphere_colors(){
                std::vector<glm::vec3> ret;
                for (unsigned int i = 0; i < sphere_faces.size(); i++){
                        ret.push_back(glm::vec3(static_cast <float> (rand()) /
                                static_cast <float> (RAND_MAX),
                                static_cast <float> (rand()) / static_cast <float>
                                        (RAND_MAX),
                                static_cast <float> (rand()) / static_cast <float>
                                        (RAND_MAX)));
                }
                return ret;
        }
}
#endif
```

Stage_control_demo.cpp:

```
#include "stdafx.h"
#include <ModelComponent.h>
#include <Gameloop.h>
#include <Component.h>
#include <iostream>
#include <glm\gtc\matrix_transform.hpp>
#include <SimpleShader.h>
#include <vector>
#include "Sphere.h"
#include "Plane.h"
#include <PhysicsComponent.h>
#include "GameObjectFactory.h"
#include "CameraControlComponent.h"
#include <fstream>
```

```
using namespace stage_control;

int _tmain(int argc, _TCHAR* argv[])
{
	//Pallot sisältävän laatikon koko
	int SCALE = 10;
	//Pallojen määrä
	int SPHERES = 5;
	int WAIT = 0;
	std::string configfile;
	std::ifstream configStream("config.ini", std::ios::in);

	//Luetaan konfiguraatiotiedosto
	if (configStream.is_open())
	{
		std::string line = "";
		std::string start, end;
		int delimiterPos;
		while (getline(configStream, line)){
			delimiterPos = line.find("=");
			//Rivi ei sisällä "="-merkkiä
			if (delimiterPos == std::string::npos){
				std::cerr << "Invalid configuration parameter " <<
					start << std::endl;
				continue;
			}
			start = line.substr(0, delimiterPos);
			end = line.substr(delimiterPos+1);
			//Parametri SCALE
			if (start == "SCALE"){
				try{
					SCALE = std::stoi(end);
					if (SCALE < 5) SCALE = 5;
				}
				catch(...){
					std::cerr << "Error parsing configuration
						parameter SCALE" << std::endl;
					continue;
				}
			}
			//Parametri SPHERES
			else if (start == "SPHERES"){
				try{
					SPHERES = std::stoi(end);
					if (SPHERES < 1) SPHERES = 1;
				}
				catch (...){
					std::cerr << "Error parsing configuration
						parameter SPHERES" << std::endl;
					continue;
				}
			}
			//Parametri WAIT
			else if (start == "WAIT"){
				try{
					WAIT = std::stoi(end);
					if (WAIT < 0) WAIT = 0;
				}
				catch (...){
					std::cerr << "Error parsing configuration
```

```
                                parameter WAIT" << std::endl;
                        continue;
                    }
                }
                //Muu parametri
                else std::cerr << "Unknown configuration parameter " <<
                    start << std::endl;
        }
        configStream.close();
}
else std::cerr << "Warning: config.ini not found, falling back to default
        parameters" << std::endl;

//Luodaan pelisilmukkaolio
Gameloop loop(std::string("Stage control engine demo"), 640, 480);
//Luodaan pelimaailma
Scene scene;

//Luodaan pallot sisältävä laatikko
//Laatikon pohja
glm::mat4 bottompos;
bottompos = glm::scale(bottompos, glm::vec3(SCALE, 1, SCALE));
bottompos = glm::translate(bottompos, glm::vec3(0, -SCALE, 0));
GameObject& bottom = GameObjectFactory::constructWall(&scene, bottompos,
        glm::vec3(SCALE, 0, SCALE));
//Laatikon katto
glm::mat4 toppos;
toppos = glm::rotate(toppos, glm::radians(180.0f), glm::vec3(1.0f, 0.0f,
        0.0f));
toppos = glm::translate(toppos, glm::vec3(0, -SCALE, 0));
toppos = glm::scale(toppos, glm::vec3(SCALE, 1, SCALE));
GameObject& top = GameObjectFactory::constructWall(&scene, toppos,
        glm::vec3(SCALE, 0, SCALE));
//Laatikon vasen seinä
glm::mat4 leftpos;
leftpos = glm::rotate(leftpos, glm::radians(90.0f), glm::vec3(0.0f, 0.0f,
        1.0f));
leftpos = glm::translate(leftpos, glm::vec3(0, -SCALE, 0));
leftpos = glm::scale(leftpos, glm::vec3(SCALE, 1, SCALE));
GameObject& left = GameObjectFactory::constructWall(&scene, leftpos,
        glm::vec3(0, SCALE, SCALE));
//Laatikon oikea seinä
glm::mat4 rightpos;
rightpos = glm::rotate(rightpos, glm::radians(-90.0f), glm::vec3(0.0f,
        0.0f, 1.0f));
rightpos = glm::translate(rightpos, glm::vec3(0, -SCALE, 0));
rightpos = glm::scale(rightpos, glm::vec3(SCALE, 1, SCALE));
GameObject& right = GameObjectFactory::constructWall(&scene, rightpos,
        glm::vec3(0, SCALE, SCALE));
//Laatikon takaseinä
glm::mat4 backpos;
backpos = glm::rotate(backpos, glm::radians(90.0f), glm::vec3(1.0f, 0.0f,
        0.0f));
backpos = glm::translate(backpos, glm::vec3(0, -SCALE, 0));
backpos = glm::scale(backpos, glm::vec3(SCALE, 1, SCALE));
GameObject& back = GameObjectFactory::constructWall(&scene, backpos,
        glm::vec3(SCALE, SCALE, 0));
//Laatikon etuseinä
glm::mat4 frontpos;
frontpos = glm::rotate(frontpos, glm::radians(-90.0f), glm::vec3(1.0f,
        0.0f, 0.0f));
```

```
	frontpos = glm::translate(frontpos, glm::vec3(0, -SCALE, 0));
	frontpos = glm::scale(frontpos, glm::vec3(SCALE, 1, SCALE));
	GameObject& front = GameObjectFactory::constructWall(&scene, frontpos,
		glm::vec3(SCALE, SCALE, 0));

	//Luodaan laatikkoon pallot
	for (int i = 0; i < SPHERES; i++){
		GameObject& sphere =
			GameObjectFactory::constructRandomSphere(&scene,
			glm::vec3(SCALE - 1, SCALE - 1, SCALE - 1), WAIT);
	}

	//Luodaan kameraolio
	GameObject& camera = scene.createObject();
	//Kameran sijainti
	Transform* camPos = new Transform(&camera);
	camPos->setMatrix(glm::translate(glm::mat4(1.0f), glm::vec3(0, 0, -SCALE *
		2)));

	glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, SCALE *
		10.0f);
	glm::mat4 View = glm::lookAt(
		glm::vec3(0, 0, -SCALE * 2),
		glm::vec3(0, 0, 0),
		glm::vec3(0, 1, 0)
	);
	//Kameran kamerakomponentti
	CameraComponent* cam = new CameraComponent(&camera, Projection, View);
	//Kameran liikuttamisen mahdollistava komponentti
	CameraControlComponent* ccc = new CameraControlComponent(&camera);

	//Asetetaan käytettävä pelialue
	loop.setActiveScene(&scene);
	//Asetetaan käytettävä kamera
	loop.setActiveCamera(cam);

	//Käynnistetään pelisilmukka
	loop.start();

	//Pelisilmukan pysähdyttyä jäädään odottamaan käyttäjän syötettä ennen
	//ohjelman sulkemista,
	//jotta käyttäjä voi lukea ruudulta suorituskykytiedot
	char c;
	std::cin >> c;

	return 0;
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

Waiter.h:

```
#ifndef WAITER_H
#define WAITER_H
```

```
#define WAITER_ID 7

#include "stdafx.h"
#include <Component.h>

namespace stage_control{
	/**Pelioliokomponentti, joka simuloi raskasta laskentaa etsimällä jokaisen
	ruudunpäivityksen aikana tietyn määrän alkulukuja*/
	class Waiter : public Component{
	public:
		/**Luo uuden Waiter-komponentin
		@param owner  Komponentin omistava peliolio
		@param limit  Se luku, johon asti alkulukuja etsitään
		*/
		Waiter(GameObject* owner, int limit) :
			Component(owner), limit(limit){}
		/**Suorittaa komponentin päivitysvaiheen laskennan eli kuluttaa
		aikaa etsimällä alkulukuja
		@param elapsedMS	Edellisestä ruudunpäivityksestä kulunut aika
		*/
		virtual void update(float elapsedMS){
			for (int i = 3; i <= limit; i++){
				for (int j = 2; j < i; j++){
					if (i % j == 0) break;
				}
			}
		}
		/** Hakee Waiter-komponentin komponenttitunnuksen
		@returns	Komponentin komponenttitunnus
		*/
		virtual int id(){ return WAITER_ID; }
	private:
		/**Se luku, johon asti tämä komponentti laskee alkulukuja jokaisen
		ruudunpäivityksen aikana*/
		int limit;
	};
}
#endif
```

Moduuli Stage_control_event:

Event.h:

```
#ifndef EVENT_H
#define EVENT_H

#include "stdafx.h"

namespace stage_control{
	/** Tapahtumajärjestelmän viestien abstrakti yliluokka*/
	struct Event{
		/** Hakee viestin tyyppitunnuksen
		@param tyyppitunnus
		*/
		virtual unsigned int getEventType() const = 0;
	};
}
#endif
```

EventChannel.h:

```
#pragma once

#ifndef EVENTCHANNEL_H
#define EVENTCHANNEL_H

#include "stdafx.h"
#include "EventHandler.h"
#include <list>

namespace stage_control{
	/** Tapahtumajärjestelmän viestejä välittävä luokka*/
	class EventChannel{
	public:
		/** Luo uuden tapahtumakanavan*/
		EventChannel(){}
		EventChannel(const EventChannel& other) = delete;
		EventChannel& operator= (const EventChannel& other) = delete;
		/** Rekisteröi olion kanavan viestien vastaanottajaksi
		@param recipient	Olio, jolle halutaan välittää kanavan viestit
		*/
		void registerRecipient(EventHandler* recipient){
			recipients.push_back(recipient);
		}
		/** Lähettää viestin kaikille kanavaan rekisteröityneille olioille
		@param e	Lähetettävä viesti
		*/
		void broadcast(const Event& e){
			for (std::list<EventHandler*>::iterator i =
				recipients.begin(); i != recipients.end(); i++){
				(*i)->handleEvent(e);
			}
		}
		/** Lähettää viestin kaikille kanavaan rekisteröityneille olioille
		paitsi viestin lähettäjälle
		@param e		Lähetettävä viesti
		@param sender Viestin lähettäjän osoite
		*/
		void broadcastOthers(const Event& e, EventHandler* sender){
			for (std::list<EventHandler*>::iterator i =
				recipients.begin(); i != recipients.end(); i++){
				if (sender != *i) (*i)->handleEvent(e);
			}
		}
	private:
		/** Lista tämän kanavan viestien vastaanottajiksi rekisteröidyistä
		olioista*/
		std::list<EventHandler*> recipients;
	};
}
#endif
```

EventHandler.h:

```
#ifndef EVENTHANDLER_H
#define EVENTHANDLER_H

#include "stdafx.h"
#include "Event.h"

namespace stage_control{
	/** Abstrakti rajapintaluokka, jonka toteuttavat oliot voivat
```

```
	rekisteröityä ottamaan vastaan viestejä tapahtumakanavilta*/
	class EventHandler{
	public:
		/** Käsittelee tapahtumajärjestelmän kautta saapuneen viestin
		@param e	Käsiteltävä viesti
		*/
		virtual void handleEvent(const Event& e) = 0;
	};
}
#endif
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN
```

Moduuli Stage_control_scene:

Component.h:

```
#pragma once
#ifndef COMPONENT_H
#define COMPONENT_H

#include "stdafx.h"
#include "GameObject.h"

namespace stage_control{
	/** Abstrakti yliluokka peliolioon liitettäville komponenteille.*/
	class Component{
	public:
		/** Luo uuden komponentin.
		HUOM: komponentti on aina luotava new:lla ja on luonnin jälkeen
		omistajaolionsa hallinnassa.
		Komponentti tuhotaan automaattisesti omistajolion tuhoutuessa.
		@param owner  Osoitin komponentin omistavaan peliolioon
		*/
		Component(GameObject* owner) : owner(owner){
			owner->addComponent(this);
		}
		/** Tuhoaa komponentin*/
		~Component(){}
		/** Palauttaa osoittimen komponentin omistavaan peliolioon
		@returns	Osoitin tämän komponentin omistajaolioon
		*/
		GameObject& getOwner(){ return *owner; }
		/** Suorittaa pelisilmukan päivitysvaiheessa tapahtuvan laskennan
		@param elapsedMS	Edellisestä pelisilmukan suorituskerrasta
					kulunut aika
		*/
		virtual void update(float elapsedMS){}
		/** Suorittaa pelisilmukan piirtovaiheessa tapahtuvan laskennan*/
		virtual void render(){}
		/** Palauttaa tämän komponentin tarkan tyypin määrittelevän
		tunnusluvun
		@returns	Tämän komponenttiolion tietityyoin tunnus
		*/
		virtual int id() = 0;
```

```
	private:
		/** Osoitin tämän komponentin omistajaolioon*/
		GameObject* owner;
	};
}
#endif
```

GameObject.h:

```
#pragma once
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include "stdafx.h"
#include <list>

namespace stage_control{
	class Component;
	/** Luokka, joka mallintaa yksittäisen peliolion.*/
	class GameObject{
		friend class Scene;
		friend class Component;
	public:
		/** Suorittaa pelisilmukan päivitysvaiheessa tapahtuvan laskennan
		@param elapsedMS	Edellisestä pelisilmukan suorituskerrasta
					kulunut aika
		*/
		void update(float elapsedMS);
		/** Suorittaa pelisilmukan piirtovaiheessa tapahtuvan laskennan
		*/
		void render();
		/** Hakee viitteen ensimmäiseen tähän olioon liitettyyn tiettyä
		tyyppiä olevaan komponenttiin
		@param id	Haettavan komponentin tyypin tunnusluku
		@returns	Osoitin haettavaan komponentiin tai nullptr, jos
				haluttua komponenttia ei ole
		*/
		Component* getComponentByID(int id);
		/** Tuhoaa peliolion*/
		~GameObject();
		/** Hakee tämän peliolion identifioivan tunnusluvun*/
		int getID(){ return id; }
	private:
		/** Tämän peliolion uniikki tunnusluku*/
		unsigned int id;
		/** Tähän peliolioon liitetyt komponentit*/
		std::list<Component*> components;
		/** Liittää tähän peliolioon uuden komponentin.
		@param comp		Osoitin liitettävään komponenttiin
		*/
		void addComponent(Component* comp);
		/** Luo uuden peliolion
		@param id	Tämän peliolion uniikki tunnusluku
		*/
		GameObject(int id) : components(), id(id){}
		/** Asettaa tämän peliolion tunnusluvun
		@param id	Uusi tunnusluku
		*/
		void setID(int id) { this->id = id; }
	};
}
```

```
#endif
```

GameObject.cpp:

```
#include "stdafx.h"
#include "GameObject.h"
#include "Component.h"

using namespace stage_control;

void GameObject::update(float elapsedMS){
	for (std::list<Component*>::iterator it = components.begin(); it !=
		components.end(); it++){
		(*it)->update(elapsedMS);
	}
}
void GameObject::render(){
	for (std::list<Component*>::iterator it = components.begin(); it !=
		components.end(); it++){
		(*it)->render();
	}
}
Component* GameObject::getComponentByID(int id){
	for (std::list<Component*>::iterator it = components.begin(); it !=
		components.end(); it++){
		if ((*it)->id() == id) return *it;
	}
	return nullptr;
}
void GameObject::addComponent(Component* comp){
	components.push_back(comp);
}
GameObject::~GameObject(){
	for (std::list<Component*>::iterator it = components.begin(); it !=
		components.end(); it++){
		delete *it;
	}
}
```

Scene.h:

```
#ifndef SCENE_H
#define SCENE_H

#include "stdafx.h"
#include "GameObject.h"
#include <list>

namespace stage_control{
	/** Pelimaailman alueen mallintava luokka*/
	class Scene{
	public:
		/** Luo uuden pelialueen*/
		Scene() : serial(0), objects(){}
		/** Tuhoaa pelialueen*/
		~Scene();
		/** Suorittaa pelisilmukan päivitysvaiheessa tapahtuvan laskennan
		@param elapsedMS	Edellisestä pelisilmukan suorituskerrasta
					kulunut aika
		*/
		void update(float elapsedMS);
		/** Suorittaa pelisilmukan piirtovaiheessa tapahtuvan laskennan*/
```

```
		void render();
		/** Luo pelimaailmaan uuden peliolion
		@returns	Viite luotuun peliolioon
		*/
		GameObject& createObject();
	private:
		/** Peliolioiden tunnusluvuista kirjaa pitävä laskuri*/
		unsigned int serial;
		/** Pelimaailmassa olevat pelioliot*/
		std::list<GameObject*> objects;
	};
}
#endif
```

Scene.cpp:

```
#include "stdafx.h"
#include "Scene.h"

using namespace stage_control;

void Scene::update(float elapsedMS){
	for (std::list<GameObject*>::iterator it = objects.begin(); it !=
		objects.end(); it++){
		(*it)->update(elapsedMS);
	}
}
void Scene::render(){
	for (std::list<GameObject*>::iterator it = objects.begin(); it !=
		objects.end(); it++){
		(*it)->render();
	}
}
GameObject& Scene::createObject(){
	GameObject* newObject = new GameObject(serial);
	serial++;
	objects.push_back(newObject);
	return *newObject;
}
Scene::~Scene(){
	for (std::list<GameObject*>::iterator it = objects.begin(); it !=
		objects.end(); it++){
		delete *it;
	}
}
```

SceneManager.h:

```
#ifndef SCENEMANAGER_H
#define SCENEMANAGER_H

#include "stdafx.h"
#include "Scene.h"
#include <Logger.h>

namespace stage_control{
	/** Abstrakti singleton-luokka, joka hallinnoi pelimaailman alueita*/
	class SceneManager{
	public:
		/** Asettaa pelin aktiivisen pelialueen
		@param scene  Osoitin uuteen aktiiviseen pelialueeseen
		*/
```

```
		virtual void setActiveScene(Scene* scene) = 0;
		/** Palauttaa pelimoottorin käynnistämisestä kuluneiden
		ruudunpäivitysten määrän
		@returns	Pelimoottorin käynnistämisestä kuluneiden
				ruudunpäivitysten määrä
		*/
		virtual unsigned int getCurrentFrame() = 0;
		/** Hakee osoittimen globaaliin alueita hallinnoivaan olioon
		@returns	Osoitin globaaliin SceneManager-singletoniin
		*/
		static SceneManager* getGlobalManager();
		/** Hakee viitteen globaaliin lokiolioon
		@returns	Viite globaaliin Logger-singletoniin
		*/
		static stage_common::Logger& getGlobalLogger();
		/** Pysäyttää ohjelman suorituksen*/
		virtual void stop() = 0;
	protected:
		/** Asettaa globaalin pelialueidenhallintaolion arvon
		@param mgr	Osoitin globaaliin SceneManager-singletoniin
		*/
		static void setGlobalManager(SceneManager* mgr);
		/** Osoitin globaaliin Logger-singletoniin*/
		static stage_common::Logger* globalLogger;
	private:
		/** Osoitin globaaliin SceneManager-singletoniin*/
		static SceneManager* globalManager;
	};
}
#endif
```

SceneManager.cpp:

```
#include "stdafx.h"
#include "SceneManager.h"

using namespace stage_control;

SceneManager* SceneManager::globalManager = nullptr;
stage_common::Logger* SceneManager::globalLogger = nullptr;

void SceneManager::setGlobalManager(SceneManager* mgr){
	globalManager = mgr;
}
SceneManager* SceneManager::getGlobalManager(){
	return globalManager;
}
stage_common::Logger& SceneManager::getGlobalLogger(){
	if (globalLogger == nullptr){
		abort();
	}
	return *globalLogger;
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"
#include <algorithm>

#define WIN32_LEAN_AND_MEAN
```

Transform.h:

```
#pragma once
#ifndef TRANSFORM_H
#define TRANSFORM_H

#define TRANSFORM_ID 1

#include "stdafx.h"
#include "Component.h"
#include <glm\glm.hpp>
#include <glm\gtx\matrix_decompose.hpp>

namespace stage_control{
	/** Peliolion sijaintia pelimaailmassa mallintava luokka*/
	class Transform : public Component{
	public:
		/** Luo uuden sijaintikomponentin ja asettaa sijainnin origoon.
		Katso oikea käyttö yliluokasta.
		@param owner  Komponentin omistava peliolio
		*/
		Transform(GameObject* owner): Component(owner){
			transform = glm::mat4();
		}
		/** Luo uuden sijaintikomponentin.
		Katso oikea käyttö yliluokasta.
		@param owner  Komponentin omistava peliolio
		@param tr            Sijainti simulaation alussa
		*/
		Transform(GameObject* owner, const glm::mat4& tr) :
			Component(owner), transform(tr){}
		/** Palauttaa komponentin mallintamaa sijaintia kuvaavan 4x4-
		matriisin
		@returns      4x4-matriisi, joka kertoo, missä pelimaailman
		              pisteessä komponentin omistava peliolio on
		*/
		glm::mat4 getMatrix() const{ return transform; }
		/** Asettaa komponentin mallintamaa sijaintia kuvaavan 4x4-
		matriisin
		@returns      4x4-matriisi, joka kertoo, missä pelimaailman
		              pisteessä komponentin omistava peliolio on
		*/
		void setMatrix(const glm::mat4& tr){
			transform = tr;
		}
		/** Hakee peliolion sijaintivektorin 3D-maailmassa
		@returns      Vektori, joka kuvaa tämän komponentin isäntäolion
		              xyz-koordinaatteja pelimaailmassa
		*/
		glm::vec3 getPosition(){
			return glm::vec3(transform[3]);
		}
		/** Siirtää pelioliota haluttuun suuntaan
		@param direction     Vektori, joka ilmoittaa miten kauas ja mihin
		                     suuntaan pelioliota halutaan siirtää
		*/
		void translate(glm::vec3 direction){
			transform = glm::translate(transform, direction);
		}
		/** Palauttaa sijaintikomponentin yksilöivän tunnusluvun
```

```
		@returns	Sijaintikomponentin tunnusluku
		*/
		virtual int id(){ return TRANSFORM_ID; }
	private:
		/** Komponentin omistavan peliolion sijaintia pelimaailmassa
		kuvaava 4x4-matriisi*/
		glm::mat4 transform;
	};
}
#endif
```

Liite 4. Monisäikeisen vertailumoottorin toteutus

Tämä liite sisältää C++11:n säikeiden avulla rinnakkaistetun Stage 11 -pelimoottoritoteutuksen ohjelmakoodin.

Moduuli Stage_11_core:

CameraComponent.h:

```
#ifndef CAMERACOMPONENT_H
#define CAMERACOMPONENT_H

#include "stdafx.h"
#include <Transform.h>
#include <Camera.h>
#include <glm\glm.hpp>
#include <GameObject.h>
#include <TaskManager.h>

#define CAMERACOMPONENT_ID 2

namespace stage_11{
	/**Komponentti, joka liittää kameran peliolioon*/
	class CameraComponent : public Component{
	public:
		/**Luo uuden kamerakomponentin oletusparametreilla
		(45 asteen fov, kuvasuhde 4:3, piirtoetäisyys 0.1-100, suunnattu
		origoon)
		Katso oikea käyttö yliluokasta
		@param owner  Viite tämän komponentin omistavaan peliolioon
		*/
		CameraComponent(GameObject& owner);
		/**Luo uuden kamerakomponentin. Katso oikea käyttö yliluokasta.
		@param owner                Viite tämän komponentin omistavaan
		                            peliolioon
		@param initialProjection    Kameran projektiomatriisi simulaation
		                            alussa
		@param initialView          Kameran näkymämatriisi simulaation
		                            alussa
		*/
		CameraComponent(GameObject& owner, glm::mat4& initialProjection,
			glm::mat4& initialView);
		/** Palauttaa kamerakomponentin komponenttitunnuksen
		@returns      Kamerakomponentin komponenttitunnus
		*/
		virtual int id(){return CAMERACOMPONENT_ID;}
		/** Valmistelee kameran ruudun piirtämistä varten
		*/
		void doRender(){
			cam.setViewMatrix(position->getMatrix());
		}
		/** Hakee osoittimen kamerakomponentin kameraolioon
		@returns      osoitin kameraolioon
		*/
		stage_common::Camera* getRawCamera(){ return &cam; }
	private:
```

```
		/** Osoitin kamerakomponentin omistavan peliolion sijaintia
		ylläpitävään olioon*/
		Transform* position;
		/** Kameraolio*/
		stage_common::Camera cam;
	};
}
#endif
```

CameraComponent.cpp:

```
#include "CameraComponent.h"

using namespace stage_11;

CameraComponent::CameraComponent(GameObject& owner) : Component(owner){
	position = (Transform*)owner.getComponentByID(TRANSFORM_ID);
	if (position == nullptr){
		LOGERR(std::string("Error: parent game object does not have a
			transform, can't initialize camera component"));
		abort();
	}
}
CameraComponent::CameraComponent(GameObject& owner, glm::mat4& initialProjection,
	glm::mat4& initialView) :
	Component(owner), cam(initialProjection, initialView){
	position = (Transform*)owner.getComponentByID(TRANSFORM_ID);
	if (position == nullptr){
		LOGERR(std::string("Error: parent game object does not have a
			transform, can't initialize camera component"));
		abort();
	}
}
```

Gameloop.h:

```
#ifndef GAMELOOP_H
#define GAMELOOP_H

#include "stdafx.h"
#include <iostream>
#include <thread>
#include <TaskPool.h>
#include <algorithm>
#include <TaskManager.h>
#include <SceneManager.h>
#include "GraphicsControlWrapper.h"
#include "CameraComponent.h"
#include <Timer.h>
#include <Input.h>

namespace stage_11{
	/** Pelisilmukan toteuttava luokka.
	Luo peli-ikkunan ja käynnistää pelin.*/
	class Gameloop : public TaskManager, public SceneManager {
	public:
		/** Luo uuden pelisilmukan
		@param windowName	Peli-ikkunan nimi
		@param xres			Peli-ikkunan sivusuuntainen resoluutio
		@param yres			Peli-ikkunan pystysuuntainen resoluutio
		@param threads		Pelimoottorin työntekijäsäikeiden määrä
		*/
```

```
		Gameloop(std::string& windowName, int xres, int yres, unsigned int
			threads = 1);
		/** Tuhoaa pelisilmukkaolion ja kaikki pelin pelialueet*/
		~Gameloop();
		/** Käynnistää pelisilmukan suorituksen*/
		void start();
		/** Suorittaa pelisilmukkaa, kunnes stop()-metodia kutsutaan*/
		void loop();
		/** Lopettaa pelin*/
		void shutdown();
		/** Pysäyttää pelin suorituksen nykyisen ruudunpäivityksen
		laskennan päätyttyä*/
		void stop();
		/** Asettaa sen kameraolion, jonka näkökulmasta pelimaailma
		kuvataan
		@param cam	Osoitin kamerakomponenttiolioon, jonka näkokulmaa
				käytetään
		*/
		void setActiveCamera(CameraComponent* cam);
		/** Hakee Gameloop-singletonin osoitteen
		@returns	Osoite globaaliin Gameloop-olioon
		*/
		static Gameloop* getMainGameloop(){	return mainLoop;}
	private:
		Gameloop(const Gameloop& other) = delete;
		Gameloop& operator= (const Gameloop& other) = delete;

		/**Globaali Gameloop-singleton*/
		static Gameloop* mainLoop;
		/** Pelimoottorin työntekijäsäikeet*/
		std::list<std::thread*> threadlist;
		/** Työntekijäsäikeiden määrä*/
		unsigned int threadcount;
		/** Onko pelin suoritus lopetettu?*/
		bool terminated = false;
		/**Grafiikkamoottoria hallinnoiva olio*/
		GraphicsControlWrapper gc;
		/**Se kameraolio, jonka näkökulmasta peli kuvataan*/
		CameraComponent* activeCam;
		/**Ajastin, joka mittaa pelisilmukan suorittamiseen kuluvaa aikaa*/
		stage_common::Timer looptimer;
		/**Ajastin, joka mittaa pelisilmukan päivitysvaiheeseen kuluvaa
		aikaa*/
		stage_common::Timer updatetimer;
		/**Ajastin, joka mittaa pelisilmukan piirtovaiheeseen kuluvaa
		aikaa*/
		stage_common::Timer rendertimer;
		/**Ajastin, joka mittaa pelisilmukan ylläpitovaiheeseen kuluvaa
		aikaa*/
		stage_common::Timer maintenancetimer;
		/**Lukko, jolla suojataan terminated-muuttujaa*/
		std::mutex stopMutex;
	};
}
#endif
```

Gameloop.cpp:

```
#include "Gameloop.h"

using namespace stage_11;
```

```
Gameloop* Gameloop::mainLoop = nullptr;

Gameloop::Gameloop(std::string& windowName, int xres, int yres, unsigned int
	threads)
	: TaskManager(threads), threadcount(threads), gc(windowName, xres, yres){
	//Sallitaan vain yksi globaali Gameloop
	if (mainLoop != nullptr){
		LOGERR("Game loop already exists, terminating");
		abort();
	}
	mainLoop = this;
}
Gameloop::~Gameloop(){
	//Tuhotaan työntekijäsäikeet
	std::for_each(threadlist.begin(), threadlist.end(), [](std::thread* t)
		{delete t; });
	//Poistetaan singleton-osoite
	mainLoop = nullptr;
}
void Gameloop::start(){
	LOGMSG("Starting Engine");
	//Luodaan työntekijäsäikeet
	for (unsigned int i = 0; i < threadcount; i++){
		threadlist.push_back(new std::thread(&TaskPool::work,
			std::ref(tp)));
	}
	loop();
}
void Gameloop::loop(){
	while (!terminated){
		looptimer.start();

		//Päivitysvaihe
		updatetimer.start();
		tp.pushTask(activeScene->update(looptimer.lastTickTime()));
		tp.waitForAllDone();
		updatetimer.stop();

		//Piirtovaihe
		rendertimer.start();
		tp.pushTask(activeScene->render());
		tp.waitForAllDone();
		gc.getController().draw(*(activeCam->getRawCamera()));
		rendertimer.stop();

		//Ylläpitovaihe
		maintenancetimer.start();
		stage_common::Input::getSingleton().update(false);
		std::unique_lock<std::mutex> lock(stopMutex);
		if (gc.getController().shouldStop()) terminated = true;
		maintenancetimer.stop();
		looptimer.stop();
	}
	shutdown();
}
void Gameloop::shutdown(){
	std::unique_lock<std::mutex> lock(stopMutex);
	terminated = true;
	//Pysäytetään säieallas
	tp.terminate();
```

```
	//Odotetaan, että työntekijäsäikeet pysähtyvät
	std::for_each(threadlist.begin(), threadlist.end(), [](std::thread* t){t
		->join(); });

	std::cout << "Total runtime: " << looptimer.totalTime() << std::endl;
	std::cout << "Total frames: " << looptimer.totalTicks() << std::endl;
	std::cout << "Average loop time: " << looptimer.averageTime() <<
		std::endl;
	std::cout << "Average fps: " << std::to_string(1000 /
		looptimer.averageTime()) << std::endl;
	std::cout << "Average update time: " << updatetimer.averageTime() <<
		std::endl;
	std::cout << "Average render time: " << rendertimer.averageTime() <<
		std::endl;
	std::cout << "Average maintenance time: " <<
		maintenancetimer.averageTime() << std::endl;
}
void Gameloop::stop(){
	std::unique_lock<std::mutex> lock(stopMutex);
	terminated = true;
}
void Gameloop::setActiveCamera(CameraComponent* cam){
	activeCam = cam;
}
```

GraphicsControlWrapper.h:

```
#ifndef GRAPHICSCONTROLWRAPPER_H
#define GRAPHICSCONTROLWRAPPER_H

#include "stdafx.h"
#include <mutex>
#include <glm\glm.hpp>
#include <TaskManager.h>
#include <GraphicsController.h>

namespace stage_11{
	/** Luokka, joka tarjoaa säieturvallisen rajapinnan grafiikkamoottorin
	hallintaluokkaan*/
	class GraphicsControlWrapper{
		friend class Gameloop;
	public:
		/** Luo uuden grafiikkamoottorin
		@param windowName	Peli-ikkunan nimi
		@param xres		Peli-ikkunan vaakaresoluutio
		@param yres		Peli-ikkunan pystyresoluutio
		*/
		GraphicsControlWrapper(std::string& windowName, int xres, int
			yres);
		/** Tuhoaa grafiikkamoottorin*/
		~GraphicsControlWrapper(){
			globalController = nullptr;
		}
		/** Lisää 3D-mallin seuraavan ruudunpäivityksen aikana
		piirrettävien mallien listalle
		@param model  Osoite piirrettävään malliolioon
		@param position	Sijainti, johon malli piirretään
		*/
		void queue(const stage_common::Model* model, const glm::mat4&
			position);
		/** Hakee osoittimen globaaliin grafiikkamoottorisingletoniin
```

```
		@returns	Osoitin globaaliin grafiikkamoottoriolioon
		*/
		static GraphicsControlWrapper& getGlobalController(){ return
			*globalController;	}
	private:
		GraphicsControlWrapper(const GraphicsControlWrapper& other) =
			delete;
		GraphicsControlWrapper& operator= (const GraphicsControlWrapper&
			other) = delete;

		/** Grafiikkamoottoriolio*/
		stage_common::GraphicsController gc;
		/** Lukko, jolla estetään useampaa säiettä muuttamasta
		piirrettävien mallien listaa samanaikaisesti*/
		std::mutex gcqueuemutex;
		/** Globaali grafikkamoottorisingleton*/
		static GraphicsControlWrapper* globalController;
		/** Hakee viitteen tämän olion sisältämään grafiikkamoottoriolioon
		@returns	Viite yksisäikeiseen grafiikkamoottoriolioon
		*/
		stage_common::GraphicsController& getController(){return gc;}
	};
}
#endif
```

GraphicsControlWrapper.cpp:

```
#include "stdafx.h"
#include "GraphicsControlWrapper.h"

using namespace stage_11;

GraphicsControlWrapper* GraphicsControlWrapper::globalController = nullptr;

GraphicsControlWrapper::GraphicsControlWrapper(std::string& windowName, int xres,
	int yres) : gc(windowName, xres, yres){
	//Sallitaan vain yksi globaali GraphicsControlWrapper
	if (globalController != nullptr){
		LOGMSG("Global graphics control threadsafety wrapper already set,
			aborting");
		abort();
	}
	globalController = this;
}
void GraphicsControlWrapper::queue(const stage_common::Model* model, const
	glm::mat4& position){
	std::unique_lock<std::mutex> lock(gcqueuemutex);
	gc.queue(model, position);
}
```

ModelComponent.h:

```
#ifndef MODELCOMPONENT_H
#define MODELCOMPONENT_H

#define MODELCOMPONENT_ID 3

#include "stdafx.h"
#include <Model.h>
#include <Component.h>
#include <Transform.h>
#include "GraphicsControlWrapper.h"
```

```
#include <GameObject.h>

namespace stage_11{
	/** Komponenttiolio, jolla liitetään peliolioon 3D-malli
	*/
	class ModelComponent : public Component{
	public:
		/** Luo uuden 3D-mallikomponentin
		Katso oikea käyttö yläluokasta
		@param owner  Viite tämän peliolion omistavaan peliolioon
		@param mod		Osoitin tämän olion 3D-malliin
		*/
		ModelComponent(GameObject& owner, stage_common::Model* mod) :
			Component(owner), mod(mod){
			tf = (Transform*)owner.getComponentByID(TRANSFORM_ID);
		}
		/** Valmistelee tämän mallin piirtämisen ruudulle*/
		virtual void doRender(){
			GraphicsControlWrapper::getGlobalController().queue(mod, tf
				->getMatrix());
		}
		/** Palauttaa mallikomponentin komponenttitunnuksen
		@returns	Mallikomponentin komponenttitunnus
		*/
		virtual int id(){
			return MODELCOMPONENT_ID;
		}
	private:
		/** Osoitin tämän komponentin 3D-malliin*/
		stage_common::Model* mod;
		/** Osoitin tämän komponentin omistavan olion sijaintia
		ylläpitävään komponenttiin*/
		Transform* tf;
	};
}
#endif
```

PhysicsComponent.h:

```
#ifndef PHYSICSCOMPONENT_H
#define PHYSICSCOMPONENT_H

#include "stdafx.h"
#include <Component.h>
#include <EventChannel.h>
#include <Collisions.h>
#include <SphereCollider.h>
#include <AABBCollider.h>
#include <glm\glm.hpp>
#include <Transform.h>
#include <GameObject.h>
#include <algorithm>

#define PHYSICSCOMPONENT_ID 4
#define PHYSICS_COLLISION_EVENT_TYPE 99001

namespace stage_11{
	/** Yksinkertainen fysiikkakomponentti, joka liittää peliolioon
	törmäyshahmon,
	antaa sille liikesuunnan ja -nopeuden ja kimmottaa sen muista
	törmäyshahmoista*/
```

```
class PhysicsComponent : public Component, public EventHandler{
	friend class StaticGeometryComponent;
public:
	/** Hakee viitteen törmäystapahtumien tapahtumakanavaan
	@returns	Viite tapahtumakanavaan
	*/
	static EventChannel& getCollisionEventChannel(){
		//Palautetaan aina sama staattinen olio
		static EventChannel channel;
		return channel;
	}
	/** Törmäyshahmon siirtymisestä ilmoittava viestitietue*/
	struct CollisionEvent : public Event{
		virtual unsigned int getEventType() const {
			return PHYSICS_COLLISION_EVENT_TYPE;
		};
		/** Viite lähettäjäolioon*/
		PhysicsComponent& sender;
		/** Edellisestä ruudunpäivityksestä kulunut aika*/
		float elapsedMS;
		CollisionEvent(PhysicsComponent& sender, float elapsedMS)
			: sender(sender), elapsedMS(elapsedMS){}
	};
	/** Luo uuden fysiikkakomponentin pallotörmäyshahmolla. Katso oikea
	käyttö yliluokasta
	@param owner		Tämän komponentin omistava peliolio
	@param radius		Törmäyshahmon säde
	@param initialV		Fysiikkaolion liikesuunta ja -nopeus
				simulaation alussa
	@param mass		Fysiikkaolion massa
	*/
	PhysicsComponent(GameObject& owner, float radius, glm::vec3
		initialV, float mass);
	/** Luo uuden fysiikkakomponentin AABB-törmäyshahmolla. Katso oikea
	käyttö yliluokasta
	@param owner		Tämän komponentin omistava peliolio
	@param siz		Törmäyshahmon koko
	@param initialV		Fysiikkaolion liikesuunta ja -nopeus
				simulaation alussa
	@param mass		Fysiikkaolion massa
	*/
	PhysicsComponent(GameObject& owner, glm::vec3 size, glm::vec3
		initialV, float mass);
	/** Tuhoaa törmäyshahmon*/
	~PhysicsComponent();
	/** Päivittää fysiikkakomponentin tilan*/
	void doUpdate(float elapsedMS);
	/** Suorittaa fysiikkaolion piirtovaiheen laskennan*/
	void doRender();
	/** Käsittelee fysiikkaolion vastaanottamat viestit
	@param ev	Vastaanotettu viesti
	*/
	void handleEvent(const Event& ev);
	/** Hakee osoittimen tämän peliolion törmäyshahmoon
	@returns	Osoitin tämän peliolion törmäyshahmoon
	*/
	stage_common::Collider* getCollider(){
		return coll;
	}
	/** Hakee tämän peliolion nopeusvektorin
	@returns	Tämän peliolion nykyinen nopeusvektori
```

```
		*/
		glm::vec3 getVelocity(){
			return velocity;
		}
		/** Palauttaa fysiikkakomponentin komponenttitunnuksen
		@returns	Fysiikkkomponentin komponenttitunnus
		*/
		virtual int id(){
			return PHYSICSCOMPONENT_ID;
		}
	private:
		/** Tämän fysiikkaolion törmäyshahmo*/
		stage_common::Collider* coll;
		/** Tämän komponentin omistavan peliolion sijaintikomponentti*/
		Transform* transform;
		/** Onko tämän komponentin tila jo päivitetty tämän
		ruudunpäivityksen aikana*/
		bool updatedThisFrame = false;
		/** Tämän fysiikkolion nykyinen liikevektori*/
		glm::vec3 velocity;
		/** Tämän fysiikkolion liikevektori edellisen sijaintipäivityksen
		jälkeen*/
		glm::vec3 oldPos;
		/** Tämän fysiikkolion massa*/
		float mass;
		/** Fysiikkolion sisäistä tilaa suojaava lukko*/
		std::mutex colliderMutex;

		/** Konstruktoreille yhteinen alustusfunktio
		@param owner  Viite tämän komponentin omistamaan peliolioon
		*/
		void setup(GameObject& owner);
		/** Päivittää fysiikkaolion sijainnin nykyisen nopeusvektorin
		perusteella
		@param elapsedMS	Edellisestä ruudunpäivityksestä kulunut aika
					millisekunteina
		*/
		void updatePosition(float elapsedMS);
		/** Synkronoi fysiikkakomponentin liikkeet sen omistavan peliolion
		sijaintikomponentin kanssa
		*/
		void commitMovement();
	};
}
#endif
```

PhysicsComponent.cpp:

```
#include "PhysicsComponent.h"

using namespace stage_11;

PhysicsComponent::PhysicsComponent(GameObject& owner, float radius, glm::vec3
	initialV, float mass) :
	Component(owner), velocity(initialV), mass(mass){
	setup(owner);
	coll = new stage_common::SphereCollider(radius, transform->getPosition());
}
PhysicsComponent::PhysicsComponent(GameObject& owner, glm::vec3 size, glm::vec3
	initialV, float mass) :
	Component(owner), velocity(initialV), mass(mass){
```

```
	setup(owner);
	coll = new stage_common::AABBCollider(size, transform->getPosition());
}
PhysicsComponent::~PhysicsComponent(){
	delete coll;
}
void PhysicsComponent::doUpdate(float elapsedMS){
	updatePosition(elapsedMS);
	//Lukulukitaan tapahtumakanava
	getCollisionEventChannel().readLock();
	CollisionEvent e(*this, elapsedMS);
	//Ilmoitetaan muuttuneesta sijainnista
	getCollisionEventChannel().broadcastOthers(e, this);
	//Vapautetaan lukko
	getCollisionEventChannel().readRelease();
}
void PhysicsComponent::doRender(){
	updatedThisFrame = false;
}
void PhysicsComponent::handleEvent(const Event& ev){
	//Käsitellään vain törmäysviestit
	if (ev.getEventType() != PHYSICS_COLLISION_EVENT_TYPE) return;
	const CollisionEvent& collEv = (const CollisionEvent&)ev;
	//Jos tilaa ei vielä ole päivitetty tämän ruudunpäivityksen aikana,
	//päivitetään se nyt
	updatePosition(collEv.elapsedMS);

	std::unique_lock<std::mutex> ownlock(colliderMutex, std::defer_lock);
	std::unique_lock<std::mutex> otherlock(collEv.sender.colliderMutex,
		std::defer_lock);
	//Lukitaan oma ja toisen fysiikkaolion lukko atomisesti lukkiutumisen
	//estämiseksi
	std::lock(ownlock, otherlock);

	if (!coll->checkCollision(*collEv.sender.coll)) return;
	//Lasketaan törmäyksen aiheuttamat muutokset nopeuksiin ja sijaintiin
	stage_common::Collisions::collisionVelocityChange(velocity, mass,
	collEv.sender.velocity, collEv.sender.mass);
	stage_common::Collisions::backOff(*collEv.sender.coll, -1.0f *
	collEv.sender.velocity, *coll);
	//Synkronoidaan sijainnin muutos omistajaolion sijainnin kanssa
	collEv.sender.commitMovement();
}
void PhysicsComponent::setup(GameObject& owner){
	transform = (Transform*)(owner.getComponentByID(TRANSFORM_ID));
	//Kirjoituslukitaan tapahtumakanava ja lisätään sen vastaanottajalistaan
	//oma osoite
	getCollisionEventChannel().writeLock();
	getCollisionEventChannel().registerRecipient(this);
	getCollisionEventChannel().writeRelease();
}
void PhysicsComponent::updatePosition(float elapsedMS){
	std::unique_lock<std::mutex> lock(colliderMutex);
	if (updatedThisFrame) return;
	//Haetaan sijainti, päivitetään se ja tallennetaan se törmäyshahmoon ja
	//isäntäolion sijaintikomponenttiin
	oldPos = transform->getPosition();
	glm::vec3 translation = velocity * elapsedMS;
	coll->center = oldPos + translation;
	//Sijainninmuutos translate-metodilla, koska muut komponentit voivat
	//muuttaa sijaintia rinnakkain
```

```
	transform->translate(translation);
	oldPos = coll->center;
	updatedThisFrame = true;
}
void PhysicsComponent::commitMovement(){
	glm::vec3 translation = coll->center - oldPos;
	transform->translate(translation);
	oldPos = coll->center;
}
```

StaticGeometryComponent.h:

```
#ifndef STATICGEOMETRYCOMPONENT_H
#define STATICGEOMETRYCOMPONENT_H

#include "stdafx.h"
#include "PhysicsComponent.h"

#define STATICGEOMETRYCOMPONENT_ID 5

namespace stage_11{
	/** Pelioliokomponentti, joka mallintaa staattista, liikkumatonta
	törmäyshahmoa, kuten pelimaailman maastoa
	tai seiniä.*/
	class StaticGeometryComponent : public Component, public EventHandler{
	public:
		/** Luo uuden staattisen törmäyshahmon, joka käyttää
		pallotörmäyshahmoa
		@param owner  Komponentin omistava peliolio
		@param radius Törmäyshahmon säde
		*/
		StaticGeometryComponent(GameObject& owner, float radius);
		/** Luo uuden staattisen törmäyshahmon, joka käyttää AABB-
		törmäyshahmoa
		@param owner  Komponentin omistava peliolio
		@param radius Törmäyshahmon koko
		*/
		StaticGeometryComponent(GameObject& owner, glm::vec3 size);
		/** Palauttaa staattisen törmäyshahmokomponentin
		komponenttitunnuksen
		@returns      Komponentin komponenttitunnus
		*/
		virtual int id(){
			return STATICGEOMETRYCOMPONENT_ID;
		}
		/** Päivittää komponentin tilan
		@param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
		*/
		virtual void doUpdate();
		/** Käsittelee olion vastaanottamat tapahtumaviestit
		@param ev     Käsiteltävä viesti
		*/
		virtual void handleEvent(const Event& ev);
	private:
		/** Komponentin törmäyshahmo*/
		stage_common::Collider* coll;
		/** Komponentin omistavan peliolion sijaintikomponentti*/
		Transform* transform;
		/** Komponentin sisäistä tilaa suojaava lukko*/
		std::mutex updatemutex;
		/** Konstruktoreille yhteinen alustusfunktio
```

```
		@param owner  Viite tämän komponentin omistamaan peliolioon
		*/
		void setup(GameObject& owner);
	};
}
#endif
```

StaticGeometryComponent.cpp:

```
#include "StaticGeometryComponent.h"

using namespace stage_11;

StaticGeometryComponent::StaticGeometryComponent(GameObject& owner, float radius)
	: Component(owner){
	setup(owner);
	coll = new stage_common::SphereCollider(radius, transform->getPosition());
}
StaticGeometryComponent::StaticGeometryComponent(GameObject& owner, glm::vec3
	size) : Component(owner){
	setup(owner);
	coll = new stage_common::AABBCollider(size, transform->getPosition());
}
void StaticGeometryComponent::doUpdate(){
	std::unique_lock<std::mutex> lock(updatemutex);
	//Haetaan nykyinen sijainti, koska muut komponentit voivat muuttaa sitä
	coll->center = transform->getPosition();
}
void StaticGeometryComponent::handleEvent(const Event& ev){
	//Käsitellään vain törmäysviestit
	if (ev.getEventType() != PHYSICS_COLLISION_EVENT_TYPE) return;
	const PhysicsComponent::CollisionEvent& collEv = (const
		PhysicsComponent::CollisionEvent&)ev;

	std::unique_lock<std::mutex> ownlock(updatemutex, std::defer_lock);
	std::unique_lock<std::mutex> otherlock(collEv.sender.colliderMutex,
		std::defer_lock);
	//Lukitaan molempien törmäävien olioiden lukot atomisesti
	std::lock(ownlock, otherlock);

	if (!coll->checkCollision(*collEv.sender.coll)) return;

	//Kimmotetaan toinen kappale tästä
	collEv.sender.velocity =
		stage_common::Collisions::reflect(collEv.sender.velocity,
		coll->getCollisionNormal(*collEv.sender.getCollider(),
		collEv.sender.velocity));
	stage_common::Collisions::backOff(*collEv.sender.coll, -1.0f *
		collEv.sender.velocity, *coll);
	collEv.sender.commitMovement();
}
void StaticGeometryComponent::setup(GameObject& owner){
	transform = (Transform*)(owner.getComponentByID(TRANSFORM_ID));
	//Kirjoituslukitaan tapahtumakanava ja lisätään sen vastaanottajalistaan
	//oma osoite
	PhysicsComponent::getCollisionEventChannel().writeLock();
	PhysicsComponent::getCollisionEventChannel().registerRecipient(this);
	PhysicsComponent::getCollisionEventChannel().writeRelease();
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

Moduuli Stage_11_demo:

CameraControlComponent.h:

```
#pragma once

#include "stdafx.h"
#ifndef CAMERACONTROLCOMPONENT_H
#define CAMERACONTROLCOMPONENT_H

#define CAMERACONTROLCOMPONENT_ID 6
#define CAMERASPEED 0.025f

#include <Component.h>
#include <Transform.h>
#include <Input.h>
#include <GLFW\glfw3.h>
#include <SceneManager.h>
#include <glm\gtx\quaternion.hpp>
#include <Gameloop.h>

namespace stage_11{
	/** Peliolioon liitettävä komponentti, joka mahdollistaa peliolion
	liikuttamisen näppäimistöllä.
	Suunniteltu kameran liikuttamiseen*/
	class CameraControlComponent : public Component{
	public:
		/** Luo uuden ohjauskomponentin
		@param owner  Komponentin omistava olio
		*/
		CameraControlComponent(GameObject& owner) : Component(owner){
			tr = (Transform*)(owner.getComponentByID(TRANSFORM_ID));
			//Rekisteröidään tarkkailtavat näppäimet
			stage_common::Input& in =
				stage_common::Input::getSingleton();
			in.registerKey(GLFW_KEY_W);
			in.registerKey(GLFW_KEY_S);
			in.registerKey(GLFW_KEY_A);
			in.registerKey(GLFW_KEY_D);
			in.registerKey(GLFW_KEY_R);
			in.registerKey(GLFW_KEY_F);
			in.registerKey(GLFW_KEY_ESCAPE);
		}
		/** Päivittää komponentin tilan
		@param elapsedMS	Edellisestä ruudunpäivityksestä kulunut aika
		*/
		void doUpdate(float elapsedMS){
			stage_common::Input& in =
				stage_common::Input::getSingleton();
			//Suunta, johon liikutaan
			glm::vec3 movement;
```

```
                        //Tarkistetaan mitä näppäimiä on painettu
                        if (in.getKeyDown(GLFW_KEY_W)) movement.z += CAMERASPEED *
                                elapsedMS;
                        if (in.getKeyDown(GLFW_KEY_S)) movement.z -= CAMERASPEED  *
                                elapsedMS;
                        if (in.getKeyDown(GLFW_KEY_A)) movement.x += CAMERASPEED  *
                                elapsedMS;
                        if (in.getKeyDown(GLFW_KEY_D)) movement.x -= CAMERASPEED  *
                                elapsedMS;
                        if (in.getKeyDown(GLFW_KEY_F)) movement.y += CAMERASPEED  *
                                elapsedMS;
                        if (in.getKeyDown(GLFW_KEY_R)) movement.y -= CAMERASPEED  *
                                elapsedMS;
                        //Siirretään isäntäoliota
                        tr->translate(movement);
                        //Suljetaan peli, jos painetaan escapea
                        if (in.getKeyDown(GLFW_KEY_ESCAPE))
                                Gameloop::getMainGameloop()->stop();
                }
                /** Hakee tämän komponentin komponenttitunnuksen
                @returns        Komponentin tunnus
                */
                int id(){
                        return CAMERACONTROLCOMPONENT_ID;
                }
        private:
                /** Isäntäolion sijainnista kirjaa pitävä olio*/
                Transform* tr;
        };
}
#endif
```

GameObjectFactory.h:

```
#ifndef GAMEOBJECTFACTORY_H
#define GAMEOBJECTFACTORY_H

#include "stdafx.h"
#include "Plane.h"
#include "Sphere.h"
#include "Waiter.h"
#include <Scene.h>
#include <glm\glm.hpp>
#include <PhysicsComponent.h>
#include <StaticGeometryComponent.h>
#include <ModelComponent.h>
#include <SimpleShader.h>

namespace stage_11{
        /** Luokka, jota käytetään demo-ohjelman peliolioiden luomiseen
        */
        class GameObjectFactory{
        public:
                /** Luo satunnaiseen paikkaan pallon, joka liikkuu satunnaiseen
                suuntaan
                @param sc                       Pelialue, jolle pallo luodaan
                @param maxCoordinates           Luotavan pallon maksimietäisyys origosta
                @param waitLimit                Mihin lukuun asti pallo laskee
                                                alkulukuja jokaisen ruudunpäivityksen
                                                aikana
                @returns                        Viite luotuun peliolioon
```

```
	*/
	static GameObject& constructRandomSphere(Scene* sc, glm::vec3
		maxCoordinates, int waitLimit){
		//Luodaan peliolio
		GameObject& obj = sc->createObject();
		//Arvotaan peliolion alkusijainti
		glm::vec3 translation(randomFloat(-maxCoordinates.x,
			maxCoordinates.x),
			randomFloat(-maxCoordinates.y, maxCoordinates.y),
			randomFloat(-maxCoordinates.z, maxCoordinates.z));
		glm::mat4 transform = glm::translate(glm::mat4(1.0f),
			translation);
		//Luodaan pelioliolle sijaintikomponentti
		Transform* tf = new Transform(obj, transform);
		//Kiinnitetään peliolioon 3D-malli
		ModelComponent* mod = new ModelComponent(obj,
			&(getSingleton().mod_sphere));
		//Arvotaan peliolion alkunopeus
		glm::vec3 velocity(randomFloat(-0.01f, 0.01f), randomFloat(-
			0.01f, 0.01f), randomFloat(-0.01f, 0.01f));
		//Luodaan pelioliolle fysiikkakomponentti
		PhysicsComponent* pc = new PhysicsComponent(obj, 1.0f,
			velocity, 1.0f);
		if (waitLimit > 0) Waiter* w = new Waiter(obj, waitLimit);
		return obj;
	}
	/** Luo pelimaailmaan seinän eli litteän, staattisen törmäyspinnan
	@param sc			Pelialue, johon seinä luodaan
	@param transform	Seinän keskipisteen 3D-sijainti
	@param size			Seinän koko
	@returns			Viite luotuun peliolioon
	*/
	static GameObject& constructWall(Scene* sc, const glm::mat4&
		transform, glm::vec3 size){
		//Luodaan uusi peliolio
		GameObject& obj = sc->createObject();
		//Luodaan pelioliolle sijaintikomponentti
		Transform* tf = new Transform(obj, transform);
		//Kiinnitetään peliolioon 3D-malli
		ModelComponent* mod = new ModelComponent(obj,
			&(getSingleton().mod_plane));
		//Luodaan pelioliolle liikkumaton törmäyshahmo
		StaticGeometryComponent* sgc = new
			StaticGeometryComponent(obj, size);
		return obj;
	}
private:
	/** Peliolioiden käyttämä sävytinohjelma*/
	stage_common::SimpleShader ss;
	/** Pallon 3D-malli*/
	stage_common::Model mod_sphere;
	/** Seinän 3D-malli*/
	stage_common::Model mod_plane;
	/**Hakee viitteen globaaliin singleton-olioon, joka luo 3D-mallit
	ja niiden sävytinohjelman sekä pitää ne muistissa
	@returns	Viite singleton-olioon
	*/
	static GameObjectFactory& getSingleton(){
		//Luodaan singleton ensimmäisellä suorituskerralla
		static GameObjectFactory gof;
		return gof;
```

```
		}
		/**Arpoo liukuluvun kahden liukuluvun väliltä
		@param start  Arvottavan luvun alaraja
		@param end		Arvottavan luvun yläraja
		@returns		Satunnainen luku ala- ja ylärajojen väliltä
		*/
		static float randomFloat(float start, float end){
			float random = ((float)rand()) / (float)RAND_MAX;
			return start + (end - start) * random;
		}
		/** Luo uuden GameObjectFactory-olion, eli käytännössä lataa
		muistiin peliolioiden luomisessa tarvittavat resurssit
		*/
		GameObjectFactory() : mod_sphere(generate_sphere_vertices(),
			generate_sphere_colors(), &ss),
			mod_plane(generate_plane_vertices(),
			generate_plane_colors(), &ss){
		}
	};
}
#endif
```

Plane.h:

```
#ifndef PLANE_H
#define PLANE_H

#include "stdafx.h"
#include <vector>
#include <glm\glm.hpp>

/*Sisältää litteän pinnan 3D-mallin datan*/
namespace stage_11{
	/** Litteän pinnan verteksit*/
	static std::vector<glm::vec3> plane_vertices = {
		glm::vec3(-1.000000, 0.000000, 1.000000),
		glm::vec3(1.000000, 0.000000, 1.000000),
		glm::vec3(-1.000000, 0.000000, -1.000000),
		glm::vec3(1.000000, 0.000000, -1.000000)
	};
	/** Pinnan tahkot
	Jokainen kolmen numeron sarja määrittelee yhden kolmion, jonka kärjet ovat
	numeroiden ilmaisemat verteksit*/
	static std::vector<int> plane_faces = {
		2, 4, 3,
		1, 2, 3
	};
	/** Yhdistää tahkot ja verteksit yhtenäiseksi verteksilistaksi
	@returns	Litteää pintaa kuvaavan 3D-mallin verteksit
	*/
	static std::vector<glm::vec3> generate_plane_vertices(){
		std::vector<glm::vec3> ret;
		//Jokaista tahkoa kohden: lisää listaan järjestyksessä tahkon
		//verteksit
		for (unsigned int i = 0; i < plane_faces.size(); i++){
			ret.push_back(plane_vertices[plane_faces[i] - 1]);
		}
		return ret;
	}
	/** Arpoo vertekseille väriarvot
	@returns	Litteää pintaa kuvaavan 3D-mallin värit
```

```
	*/
	static std::vector<glm::vec3> generate_plane_colors(){
		std::vector<glm::vec3> ret;
		for (unsigned int i = 0; i < plane_faces.size(); i++){
			ret.push_back(glm::vec3(static_cast <float> (rand()) /
				static_cast <float> (RAND_MAX),
				static_cast <float> (rand()) / static_cast <float>
				(RAND_MAX),
				static_cast <float> (rand()) / static_cast <float>
				(RAND_MAX)));
		}
		return ret;
	}
}
#endif
```

simplefragmentshader.glsl:

```
//Pikselisävytin
#version 330 core

//Parametri: värit
in vec3 fragmentColor;
//Palautetaan lasketut värit
out vec3 color;

void main(){
    color = fragmentColor;
}
```

simplevertexshader.glsl:

```
//Verteksisävytin
#version 330 core

//1. parametri: verteksit
layout(location = 0) in vec3 vertexPosition_modelspace;
//2. parametri: värit
layout(location = 1) in vec3 vertexColor;

//Annetaan värit pikselisävyttimelle
out vec3 fragmentColor;

//Mallin sijainti suhteessa kameraan
uniform mat4 MVP;

//Lasketaan verteksin sijainti ruudulla suhteessa malliin ja annetaan värit
//pikselisävyttimelle
void main(){
    vec4 v = vec4(vertexPosition_modelspace,1);
    gl_Position = MVP * v;
       fragmentColor = vertexColor;
}
```

Sphere.h:

```
#ifndef SPHERE_H
#define SPHERE_H

#include "stdafx.h"
#include <vector>
#include <glm\glm.hpp>
```

```
/*Sisältää palloa approksimoivan 3D-mallin datan*/
namespace stage_11{
	/** Pallon verteksit*/
	static std::vector<glm::vec3> sphere_vertices = {
		glm::vec3(0.000000, -1.000000, 0.000000),
		glm::vec3(0.723607, -0.447220, 0.525725),
		glm::vec3(-0.276388, -0.447220, 0.850649),
		glm::vec3(-0.894426, -0.447216, 0.000000),
		glm::vec3(-0.276388, -0.447220, -0.850649),
		glm::vec3(0.723607, -0.447220, -0.525725),
		glm::vec3(0.276388, 0.447220, 0.850649),
		glm::vec3(-0.723607, 0.447220, 0.525725),
		glm::vec3(-0.723607, 0.447220, -0.525725),
		glm::vec3(0.276388, 0.447220, -0.850649),
		glm::vec3(0.894426, 0.447216, 0.000000),
		glm::vec3(0.000000, 1.000000, 0.000000),
		glm::vec3(-0.162456, -0.850654, 0.499995),
		glm::vec3(0.425323, -0.850654, 0.309011),
		glm::vec3(0.262869, -0.525738, 0.809012),
		glm::vec3(0.850648, -0.525736, 0.000000),
		glm::vec3(0.425323, -0.850654, -0.309011),
		glm::vec3(-0.525730, -0.850652, 0.000000),
		glm::vec3(-0.688189, -0.525736, 0.499997),
		glm::vec3(-0.162456, -0.850654, -0.499995),
		glm::vec3(-0.688189, -0.525736, -0.499997),
		glm::vec3(0.262869, -0.525738, -0.809012),
		glm::vec3(0.951058, 0.000000, 0.309013),
		glm::vec3(0.951058, 0.000000, -0.309013),
		glm::vec3(0.000000, 0.000000, 1.000000),
		glm::vec3(0.587786, 0.000000, 0.809017),
		glm::vec3(-0.951058, 0.000000, 0.309013),
		glm::vec3(-0.587786, 0.000000, 0.809017),
		glm::vec3(-0.587786, 0.000000, -0.809017),
		glm::vec3(-0.951058, 0.000000, -0.309013),
		glm::vec3(0.587786, 0.000000, -0.809017),
		glm::vec3(0.000000, 0.000000, -1.000000),
		glm::vec3(0.688189, 0.525736, 0.499997),
		glm::vec3(-0.262869, 0.525738, 0.809012),
		glm::vec3(-0.850648, 0.525736, 0.000000),
		glm::vec3(-0.262869, 0.525738, -0.809012),
		glm::vec3(0.688189, 0.525736, -0.499997),
		glm::vec3(0.162456, 0.850654, 0.499995),
		glm::vec3(0.525730, 0.850652, 0.000000),
		glm::vec3(-0.425323, 0.850654, 0.309011),
		glm::vec3(-0.425323, 0.850654, -0.309011),
		glm::vec3(0.162456, 0.850654, -0.499995)
	};
	/** Pallon tahkot
	Jokainen kolmen numeron sarja määrittelee yhden kolmion, jonka kärjet ovat
	numeroiden ilmaisemat verteksit*/
	static std::vector<int> sphere_faces = {
		1, 14, 13,
		2, 14, 16,
		1, 13, 18,
		1, 18, 20,
		1, 20, 17,
		2, 16, 23,
		3, 15, 25,
		4, 19, 27,
		5, 21, 29,
		6, 22, 31,
```

```
2, 23, 26,
3, 25, 28,
4, 27, 30,
5, 29, 32,
6, 31, 24,
7, 33, 38,
8, 34, 40,
9, 35, 41,
10, 36, 42,
11, 37, 39,
39, 42, 12,
39, 37, 42,
37, 10, 42,
42, 41, 12,
42, 36, 41,
36, 9, 41,
41, 40, 12,
41, 35, 40,
35, 8, 40,
40, 38, 12,
40, 34, 38,
34, 7, 38,
38, 39, 12,
38, 33, 39,
33, 11, 39,
24, 37, 11,
24, 31, 37,
31, 10, 37,
32, 36, 10,
32, 29, 36,
29, 9, 36,
30, 35, 9,
30, 27, 35,
27, 8, 35,
28, 34, 8,
28, 25, 34,
25, 7, 34,
26, 33, 7,
26, 23, 33,
23, 11, 33,
31, 32, 10,
31, 22, 32,
22, 5, 32,
29, 30, 9,
29, 21, 30,
21, 4, 30,
27, 28, 8,
27, 19, 28,
19, 3, 28,
25, 26, 7,
25, 15, 26,
15, 2, 26,
23, 24, 11,
23, 16, 24,
16, 6, 24,
17, 22, 6,
17, 20, 22,
20, 5, 22,
20, 21, 5,
20, 18, 21,
18, 4, 21,
```

```
			18, 19, 4,
			18, 13, 19,
			13, 3, 19,
			16, 17, 6,
			16, 14, 17,
			14, 1, 17,
			13, 15, 3,
			13, 14, 15,
			14, 2, 15
		};
		/** Yhdistää tahkot ja verteksit yhtenäiseksi verteksilistaksi
		@returns	Palloa kuvaavan 3D-mallin verteksit
		*/
		static std::vector<glm::vec3> generate_sphere_vertices(){
			std::vector<glm::vec3> ret;
			for (unsigned int i = 0; i < sphere_faces.size(); i++){
				ret.push_back(sphere_vertices[sphere_faces[i] - 1]);
			}
			return ret;
		}
		/** Arpoo vertekseille väriarvot
		@returns	Litteää pintaa kuvaavan 3D-mallin värit
		*/
		static std::vector<glm::vec3> generate_sphere_colors(){
			std::vector<glm::vec3> ret;
			for (unsigned int i = 0; i < sphere_faces.size(); i++){
				ret.push_back(glm::vec3(static_cast <float> (rand()) /
					static_cast <float> (RAND_MAX),
					static_cast <float> (rand()) / static_cast <float>
					(RAND_MAX),
					static_cast <float> (rand()) / static_cast <float>
					(RAND_MAX)));
			}
			return ret;
		}
}
#endif
```

Stage_11_demo.cpp:

```
#include "stdafx.h"
#include <ModelComponent.h>
#include <Gameloop.h>
#include <Component.h>
#include <iostream>
#include <glm\gtc\matrix_transform.hpp>
#include <SimpleShader.h>
#include <vector>
#include "Sphere.h"
#include "Plane.h"
#include <PhysicsComponent.h>
#include "GameObjectFactory.h"
#include "CameraControlComponent.h"
#include <fstream>

using namespace stage_11;


int _tmain(int argc, _TCHAR* argv[])
{
	//Pallot sisältävän laatikon koko
```

```
int SCALE = 10;
//Pallojen määrä
int SPHERES = 5;
//Säikeiden määrä
int THREADS = 16;
//Kuinka raskasta laskentaa simuloidaan
int WAIT = 0;
std::string configfile;
std::ifstream configStream("config.ini", std::ios::in);

//Luetaan konfiguraatiotiedosto
if (configStream.is_open())
{
	std::string line = "";
	std::string start, end;
	int delimiterPos;
	while (getline(configStream, line)){
		delimiterPos = line.find("=");
		//Rivi ei sisällä "="-merkkiä
		if (delimiterPos == std::string::npos){
			std::cerr << "Invalid configuration parameter " <<
				start << std::endl;
			continue;
		}
		start = line.substr(0, delimiterPos);
		end = line.substr(delimiterPos + 1);
		//Parametri SCALE
		if (start == "SCALE"){
			try{
				SCALE = std::stoi(end);
				if (SCALE < 5) SCALE = 5;
			}
			catch (...){
				std::cerr << "Error parsing configuration
					parameter SCALE" << std::endl;
				continue;
			}
		}
		//Parametri SPHERES
		else if (start == "SPHERES"){
			try{
				SPHERES = std::stoi(end);
				if (SPHERES < 1) SPHERES = 1;
			}
			catch (...){
				std::cerr << "Error parsing configuration
					parameter SPHERES" << std::endl;
				continue;
			}
		}
		//Parametri THREADS
		else if (start == "THREADS"){
			try{
				THREADS = std::stoi(end);
				if (THREADS < 1) THREADS = 1;
			}
			catch (...){
				std::cerr << "Error parsing configuration
					parameter THREADS" << std::endl;
				continue;
			}
```

```
                }
                //Parametri WAIT
                else if (start == "WAIT"){
                        try{
                                WAIT = std::stod(end);
                                if (WAIT < 0) WAIT = 0;
                                std::cout << "Parsed wait value: " <<
                                        std::to_string(WAIT) << std::endl;
                        }
                        catch (...){
                                std::cerr << "Error parsing configuration
                                        parameter WAIT" << std::endl;
                                continue;
                        }
                }
                //Muu parametri
                else std::cerr << "Unknown configuration parameter " <<
                        start << std::endl;
        }
        configStream.close();
}
else std::cerr << "Warning: config.ini not found, falling back to default
        parameters" << std::endl;

//Luodaan pelisilmukkaolio
Gameloop loop(std::string("Stage 11 engine demo"), 640, 480, THREADS);
//Luodaan pelimaailma
Scene& scene = loop.createScene();

//Luodaan pallot sisältävä laatikko
//Laatikon pohja
glm::mat4 bottompos;
bottompos = glm::scale(bottompos, glm::vec3(SCALE, 1, SCALE));
bottompos = glm::translate(bottompos, glm::vec3(0, -SCALE, 0));
GameObject& bottom = GameObjectFactory::constructWall(&scene, bottompos,
        glm::vec3(SCALE, 0, SCALE));
//Laatikon katto
glm::mat4 toppos;
toppos = glm::rotate(toppos, glm::radians(180.0f), glm::vec3(1.0f, 0.0f,
        0.0f));
toppos = glm::translate(toppos, glm::vec3(0, -SCALE, 0));
toppos = glm::scale(toppos, glm::vec3(SCALE, 1, SCALE));
GameObject& top = GameObjectFactory::constructWall(&scene, toppos,
        glm::vec3(SCALE, 0, SCALE));
//Laatikon vasen seinä
glm::mat4 leftpos;
leftpos = glm::rotate(leftpos, glm::radians(90.0f), glm::vec3(0.0f, 0.0f,
        1.0f));
leftpos = glm::translate(leftpos, glm::vec3(0, -SCALE, 0));
leftpos = glm::scale(leftpos, glm::vec3(SCALE, 1, SCALE));
GameObject& left = GameObjectFactory::constructWall(&scene, leftpos,
        glm::vec3(0, SCALE, SCALE));
//Laatikon oikea seinä
glm::mat4 rightpos;
rightpos = glm::rotate(rightpos, glm::radians(-90.0f), glm::vec3(0.0f,
        0.0f, 1.0f));
rightpos = glm::translate(rightpos, glm::vec3(0, -SCALE, 0));
rightpos = glm::scale(rightpos, glm::vec3(SCALE, 1, SCALE));
GameObject& right = GameObjectFactory::constructWall(&scene, rightpos,
        glm::vec3(0, SCALE, SCALE));
//Laatikon takaseinä
```

```
	glm::mat4 backpos;
	backpos = glm::rotate(backpos, glm::radians(90.0f), glm::vec3(1.0f, 0.0f,
		0.0f));
	backpos = glm::translate(backpos, glm::vec3(0, -SCALE, 0));
	backpos = glm::scale(backpos, glm::vec3(SCALE, 1, SCALE));
	GameObject& back = GameObjectFactory::constructWall(&scene, backpos,
		glm::vec3(SCALE, SCALE, 0));
	//Laatikon etuseinä
	glm::mat4 frontpos;
	frontpos = glm::rotate(frontpos, glm::radians(-90.0f), glm::vec3(1.0f,
		0.0f, 0.0f));
	frontpos = glm::translate(frontpos, glm::vec3(0, -SCALE, 0));
	frontpos = glm::scale(frontpos, glm::vec3(SCALE, 1, SCALE));
	GameObject& front = GameObjectFactory::constructWall(&scene, frontpos,
		glm::vec3(SCALE, SCALE, 0));

	//Luodaan laatikkoon pallot
	for (int i = 0; i < SPHERES; i++){
		GameObject& sphere =
			GameObjectFactory::constructRandomSphere(&scene,
			glm::vec3(SCALE - 1, SCALE - 1, SCALE - 1), WAIT);
	}

	//Luodaan kameraolio
	GameObject& camera = scene.createObject();
	//Kameran sijainti
	Transform* camPos = new Transform(camera);
	camPos->setMatrix(glm::translate(glm::mat4(1.0f), glm::vec3(0, 0, -SCALE *
		2)));

	glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, SCALE *
		10.0f);
	glm::mat4 View = glm::lookAt(
		glm::vec3(0, 0, -SCALE * 2),
		glm::vec3(0, 0, 0),
		glm::vec3(0, 1, 0)
		);
	//Kameran kamerakomponentti
	CameraComponent* cam = new CameraComponent(camera, Projection, View);
	//Kameran liikuttamisen mahdollistava komponentti
	CameraControlComponent* ccc = new CameraControlComponent(camera);

	//Asetetaan käytettävä pelialue
	loop.setActiveScene(0);
	//Asetetaan käytettävä kamera
	loop.setActiveCamera(cam);

	//Käynnistetään pelisilmukka
	loop.start();

	//Pelisilmukan pysähdyttyä jäädään odottamaan käyttäjän syötettä ennen
	//ohjelman sulkemista,
	//jotta käyttäjä voi lukea ruudulta suorituskykytiedot
	char c;
	std::cin >> c;

	return 0;
}
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

Waiter.h:

```
#ifndef WAITER_H
#define WAITER_H

#define WAITER_ID 7

#include "stdafx.h"
#include <Component.h>

namespace stage_11{
	/** Pelioliokomponentti, joka simuloi raskasta, hyvin rinnakkaistuvaa
	laskentaa etsimällä alkulukuja*/
	class Waiter : public Component{
	public:
		/** Luo uuden Waiter-komponentin. Katso oikea käyttö yliluokasta.
		@param owner  Viite tämän komponentin omistavaan peliolioon
		@param limit  Se luku, jota pienemmät alkukuvut etsitään joka
		ruudunpäivityksen aikana
		*/
		Waiter(GameObject& owner, int limit): Component(owner),
			limit(limit){}
		/** Laskee alkulukuja tiettyyn raja-arvoon asti
		@param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
		*/
		virtual void doUpdate(float elapsedMS){
			for (int i = 3; i <= limit; i++){
				for (int j = 2; j < i; j++){
					if (i % j == 0) break; //alkuluku löydetty
				}
			}
		}
		/** Palauttaa Waiter-komponentin komponenttitunnuksen
		@returns      Waiter-komponentin tunnusluku
		*/
		virtual int id(){ return WAITER_ID; }
	private:
		/** Se luku, jota pienemmät alkuluvut etsitään jokaisen
		ruudunpäivityksen aikana*/
		int limit;
	};
}
#endif
```

Moduuli Stage_11_event:

Event.h:

```
#ifndef EVENT_H
#define EVENT_H

#include "stdafx.h"

namespace stage_11{
```

```
	/** Tapahtumajärjestelmän viestien abstrakti yliluokka*/
	struct Event{
		/** Hakee viestin tyyppitunnuksen
		@param tyyppitunnus
		*/
		virtual unsigned int getEventType() const = 0;
	};
}
#endif
```

EventChannel.h:

```
#ifndef EVENTCHANNEL_H
#define EVENTCHANNEL_H

#include "stdafx.h"
#include <list>
#include "EventHandler.h"
#include <TaskManager.h>
#include <mutex>

namespace stage_11{
	/** Mallintaa tapahtumakanavaa, joka mahdollistaa viestien lähettämisen
	joukolle peliolioita*/
	class EventChannel{
	public:
		/** Luo uuden tapahtumakanavan*/
		EventChannel(){}
		/** Lisää olion tapahtumakanavan vastaanottajalistalle
		HUOM: kanava on lukittava writeLock-metodilla ennen tämän metodin
		kutsumista
		@param recipient	Osoitin vastaanottajalistalle lisättävään
					olioon
		*/
		void registerRecipient(EventHandler* recipient);
		/** Lähettää viestin kaikille vastaanottajalistan olioille yhtä
		lukuunottamatta
		HUOM: kanava on lukittava readLock-metodilla ennen tämän metodin
		kutsumista
		@param e		Lähetettävä viesti
		@param sender Se olio, jolle viestiä ei lähetetä
		*/
		template <class EventType>
		void broadcastOthers(EventType e, EventHandler* sender){
			std::list<Task*> tasks;
			for (std::list<EventHandler*>::iterator i =
				recipients.begin(); i != recipients.end(); i++){
				//Luodaan uusi työtehtävä jokaiselle viestille
				if (*i != sender) tasks.push_back(new
					Forward<EventType>(e, *i));
			}
			//Lisätään tehtävät työlistaan
			TaskManager::pushTaskList(tasks);
		}
		/** Lähettää viestin kaikille vastaanottajalistan olioille
		HUOM: kanava on lukittava readLock-metodilla ennen tämän metodin
		kutsumista
		@param e	Lähetettävä viesti
		*/
		template <class EventType>
		void broadcast(EventType e){
```

```
                broadcastOthers<EventType>(e, nullptr);
        }
        /** Lukulukitsee tapahtumakanavan
        Tämän metodin kutsumisen jälkeen kanavan kautta voidaan
        turvallisesti lähettää viestejä
        Lukko on vapautettava jälkeenpäin readRelease-metodilla*/
        void readLock();
        /** Vapauttaa kanavan lukulukon*/
        void readRelease();
        /** Kirjoituslukitsee tapahtumakanavan
        Tämän metodin kutsumisen jälkeen kanavalle voidaan turvallisesti
        lisätä kuuntelijoita
        Lukko on vapautettava jälkeenpäin writeRelease-metodilla*/
        void writeLock();
        /** Vapauttaa kanavan kirjoituslukon*/
        void writeRelease();
    private:
        EventChannel(const EventChannel& other) = delete;
        EventChannel& operator= (const EventChannel& other) = delete;

        /** Lista tämän tapahtumakanavan viestien vastaanottajiksi
        rekisteröityneistä olioista*/
        std::list<EventHandler*> recipients;
        /** Tällä hetkellä vastaanottajalistaa lukevien olioiden
        lukumäärä*/
        int readers = 0;
        /** Kirjoitetaanko vastaanottajalistaan tällä hetkellä*/
        bool writing = false;
        /** Lukko, jolla suojataan pääsyä lukemaan tai kirjoittamaan
        vastaanottajalistaan*/
        std::mutex entry;
        /** Odottavat säikeet herätetään, kun vastaanottajalistaa voi
        lukea*/
        std::condition_variable canRead;
        /** Odottava säie herätetään, kun vastaanottajalistaan voi
        kirjoittaa*/
        std::condition_variable canWrite;

        /** Työtehtävä, joka käsittelee yhden kanavan kautta lähetetyn
        viestin*/
        template <class EventType>
        class Forward : public Task{
        public:
            /** Luo uuden viestityön
            @param e            Lähetettävä viesti
            @param handler      Viestin käsittelevä olio
            */
            Forward(EventType e, EventHandler* handler) : e(e),
                handler(handler){}
            TASK_EXECUTE{
                handler->handleEvent(e);
            }
        private:
            /** Lähetetty viesti*/
            EventType e;
            /** Viestin vastaanottaja*/
            EventHandler* handler;
        };
    };
}
#endif
```

EventChannel.cpp:

```
#include "EventChannel.h"

using namespace stage_11;

void EventChannel::registerRecipient(EventHandler* recipient){
	recipients.push_back(recipient);
}
void EventChannel::readLock(){
	std::unique_lock<std::mutex> lock(entry);
	//Odotetaan, kunnes kirjoittajia ei ole
	if (writing) canRead.wait(lock, [this]{return !writing; });
	readers++;
}
void EventChannel::readRelease(){
	std::unique_lock<std::mutex> lock(entry);
	readers--;
	//Herätetään kirjoittaja, jos ei lukijoita
	if (readers < 1) canWrite.notify_one();
}
void EventChannel::writeLock(){
	std::unique_lock<std::mutex> lock(entry);
	//Odotetaan, kunnes kirjoittajia ja lukijoita ei ole
	if (writing || readers > 0) canWrite.wait(lock, [this]{return !writing &&
		readers < 1; });
	writing = true;
}
void EventChannel::writeRelease(){
	std::unique_lock<std::mutex> lock(entry);
	writing = false;
	//Herätetään yksi kirjoittaja ja kaikki lukijat; nopein saa lukon
	canWrite.notify_one();
	canRead.notify_all();
}
```

EventHandler.h:

```
#ifndef EVENTHANDLER_H
#define EVENTHANDLER_H

#include "stdafx.h"
#include "Event.h"

namespace stage_11{
	/** Abstrakti rajapintaluokka, jonka toteuttavat oliot voivat
	rekisteröityä ottamaan vastaan viestejä tapahtumakanavilta*/
	class EventHandler{
	public:
		/** Käsittelee tapahtumajärjestelmän kautta saapuneen viestin
		@param e	Käsiteltävä viesti
		*/
		virtual void handleEvent(const Event& e) = 0;
	};
}
#endif
```

stdafx.h:

```
#pragma once
```

```
#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

Moduuli Stage_11_scene:

Component.h:

```
#ifndef COMPONENT_H
#define COMPONENT_H

#include "stdafx.h"
#include <Task.h>

namespace stage_11{
	class GameObject;
	/** Yliluokka pelioliokomponenteille. Komponentit liitetään peliolioihin
	ja ne suorittavat laskentaa pelimoottorin päivitys- ja piirtovaiheissa.*/
	class Component{
	public:
		/** Suorittaa tämän komponentin päivitysvaiheen laskennan.
		@param elapsedMS	Edellisestä ruudunpäivityksestä kulunut aika
		*/
		virtual void doUpdate(float elapsedMS){}
		/** Suorittaa tämän komponentin piirtovaiheen laskennan.
		*/
		virtual void doRender(){}
		/** Palauttaa tunnusluvun, jonka avulla tunnistetaan tämän
		komponentin tyyppi
		@returns	Tämän komponentin tyyppitunnus
		*/
		virtual int id() = 0;
		/** Luo uuden komponentin. Luonnin jälkeen tämän komponentin
		omistava olio hallinnoi sen elinkaarta,
		joten komponentti on aina luotava new:lla.
		@param owner  Tämän komponentin omistava peliolio
		*/
		Component(GameObject& owner);
		/** Luo uuden työtehtävän, joka kutsuu tämän komponentin
		päivitysmetodia.
		@returns	Tämän komponentin päivitystyötehtävä. Tuhottava
				suorituksen jälkeen manuaalisesti deletellä.
		*/
		Task* update(float elapsedMS){
			return new Update(this, elapsedMS);
		}
		/** Luo uuden työtehtävän, joka kutsuu tämän komponentin
		piirtometodia.
		@returns	Tämän komponentin piirtotyötehtävä. Tuhottava
				suorituksen jälkeen manuaalisesti deletellä.
		*/
		Task* render(){
			return new Render(this);
		}
	protected:
		/** Työtehtäväolio, joka kutsuu komponenttiolion päivitysmetodia*/
		class Update : public Task{
		public:
			TASK_EXECUTE {
```

```
                        c->doUpdate(elapsedMS);
                };
                Update(Component*c, float elapsedMS) :c(c),
                elapsedMS(elapsedMS){}
        private:
                /** Se komponentti, jonka päivitysmetodia kutsutaan.*/
                Component* c;
                /** Edellisestä ruudunpäivityksestä kulunut aika.*/
                float elapsedMS;
        };
        /** Työtehtäväolio, joka kutsuu komponenttiolion piirtometodia*/
        class Render : public Task{
        public:
                TASK_EXECUTE{
                        c->doRender();
                }
                Render(Component*c) :c(c){}
        private:
                /** Se komponentti, jonka piirtometodia kutsutaan.*/
                Component* c;
        };
private:
        Component(const Component& other) = delete;
        Component& operator= (const Component& other) = delete;
};

}
#endif
```

Component.cpp:

```
#include "stdafx.h"
#include "Component.h"
#include "GameObject.h"

using namespace stage_11;

Component::Component(GameObject& owner){
        owner.addComponent(this);
}
```

GameObject.h:

```
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include "stdafx.h"
#include "Component.h"
#include <list>
#include <TaskManager.h>
#include <algorithm>
#include <mutex>

namespace stage_11{
        /** Peliolion mallintava olio. Peliolio liittää eri komponentteja yhteen
        yhdeksi kokonaisuudeksi*/
        class GameObject{
        public:
                /** Luo uuden työtehtäväolion, joka luo päivitystyötehtävät
                kaikille tämän peliolion komponenteille
                @param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
                @returns             Työtehtäväolio, joka päivittää tämän peliolion
```

```
                        komponentit
        */
        Task* update(float elapsedMS){
                return new Update(this, elapsedMS);
        }
        /** Luo uuden työtehtäväolion, joka luo piirtotyötehtävät kaikille
        tämän peliolion komponenteille
        @returns        Työtehtäväolio, joka piirtää tämän peliolion
                        komponentit
        */
        Task* render(){
                return new Render(this);
        }
        /** Liittää tähän peliolioon uuden komponentin.
        Liittämisen jälkeen peliolio hallinnoi komponentin elinkaarta,
        joten komponentti on aina luotava new:lla.
        @param comp     Osoitin liitettävään komponenttiin
        */
        void addComponent(Component* comp){
                std::unique_lock<std::mutex>(componentListMutex);
                components.push_back(comp);
        }
        /** Hakee osoittimen tämän peliolion omistamaan tietyntyyppiseen
        komponenttiin
        @param id       Etsittävän komponentin tyyppitunnus
        @returns        Osoitin etsittyyn komponentiin tai nullptr, jos
                        etsityn tyyppistä komponenttia ei ole
        */
        Component* getComponentByID(int id){
                for (std::list<Component*>::iterator it =
                        components.begin(); it != components.end(); it++){
                        if ((*it)->id() == id) return *it;
                }
                return nullptr;
        }
        /** Luo uuden peliolion*/
        GameObject(){}
        /** Tuhoaa peliolion ja sen komponentit*/
        ~GameObject(){
                std::unique_lock<std::mutex>(componentListMutex);
                std::for_each(components.begin(), components.end(), []
                        (Component* c){
                        delete c;
                });
        }
private:
        /** Tämän peliolion omistamat komponentit*/
        std::list<Component*> components;
        /** Komponenttilistaa suojaava lukko*/
        std::mutex componentListMutex;

        GameObject(const GameObject& other) = delete;
        GameObject& operator= (const GameObject& other) = delete;

        /** Työtehtäväolio, joka päivittää peliolion komponenttien tilan*/
        class Update : public Task{
        public:
                TASK_EXECUTE{
                        std::unique_lock<std::mutex>(go->componentListMutex);
                        std::list<Task*> tasks;
                        std::for_each(go->components.begin(), go
```

```
                                ->components.end(), [this, &tasks](Component*
                                c){
                                tasks.push_back(c->update(elapsedMS));
                        });
                        //Lisätään kaikki komponenttipäivitykset kerralla
                        TaskManager::pushTaskList(tasks);
                };
                Update(GameObject* go, float elapsedMS) :go(go),
                        elapsedMS(elapsedMS){}
        private:
                /** Se peliolio, jonka komponentit päivitetään*/
                GameObject* go;
                /** Edellisestä ruudunpäivityksestä kulunut aika*/
                float elapsedMS;
        };
        /** Työtehtäväolio, joka päivittää valmistelee peliolion
        komponentit piirtoa varten*/
        class Render : public Task{
        public:
                TASK_EXECUTE{
                        std::unique_lock<std::mutex>(go->componentListMutex);
                        std::list<Task*> tasks;
                        std::for_each(go->components.begin(), go
                                ->components.end(), [&tasks](Component* c){
                                tasks.push_back(c->render());
                        });
                        //Lisätään kaikki työtehtävät kerralla
                        TaskManager::pushTaskList(tasks);
                }
                Render(GameObject* go) :go(go){}
        private:
                /** Se peliolio, jonka komponentit piirretään*/
                GameObject* go;
        };
};
}
#endif
```

Scene.h:

```
#ifndef SCENE_H
#define SCENE_H

#include "stdafx.h"
#include <Task.h>
#include <list>
#include "GameObject.h"

namespace stage_11{
        /** Pelialueen mallintava luokka.*/
        class Scene{
        public:
                /** Luo työtehtävän, joka päivittää pelialueen peliolioiden tilan
                @param elapsedMS     Edellisestä ruudunpäivityksestä kulunut aika
                @returns             Työtehtäväolio, joka päivittää tämän
                                     pelialueen pelioliot
                */
                Task* update(float elapsedMS){
                        return new Update(this, elapsedMS);
                }
                /** Luo työtehtävän, joka piirtää pelialueen pelioliot
```

```
        @returns      Työtehtäväolio, joka piirtää tämän pelialueen
                      pelioliot
        */
        Task* render(){
                return new Render(this);
        }
        /** Luo pelialueelle uuden peliolion
        @returns      Viite luotuun peliolioon
        */
        GameObject& createObject(){
                std::unique_lock<std::mutex>(objectListMutex);
                GameObject* obj = new GameObject();
                objects.push_back(obj);
                return *obj;
        }
        /** Luo uuden pelialueen*/
        Scene(){}
        /** Tuhoaa pelialueen ja sen sisältämät pelioliot*/
        ~Scene(){
                std::unique_lock<std::mutex>(objectListMutex);
                std::for_each(objects.begin(), objects.end(), [](GameObject*
                        go){
                        delete go;
                });
        }
private:
        /** Pelialueen sisältämät pelioliot*/
        std::list<GameObject*> objects;
        /** Peliolioliostaa suojaava lukko*/
        std::mutex objectListMutex;

        Scene(const Scene& other) = delete;
        Scene& operator= (const Scene& other) = delete;

        /** Työtehtäväolio, joka päivittää pelialueen sisältämien
        peliolioiden tilan*/
        class Update : public Task{
        public:
                Update(Scene* sc, float elapsedMS) : sc(sc),
                        elapsedMS(elapsedMS){}
                void operator()(){
                        std::unique_lock<std::mutex>(sc->objectListMutex);
                        std::list<Task*> tasks;
                        std::for_each(sc->objects.begin(), sc->objects.end(),
                                [this, &tasks](GameObject* go){
                                tasks.push_back(go->update(elapsedMS));
                        });
                        TaskManager::pushTaskList(tasks);
                }
        private:
                /** Se pelialue, jonka oliot päivitetään*/
                Scene* sc;
                /** Edellisestä ruudunpäivityksestä kulunut aika*/
                float elapsedMS;
        };
        /** Työtehtäväolio, joka piirtää ruudulle pelialueen sisältämät
        pelioliot*/
        class Render : public Task{
        public:
                Render(Scene* sc) : sc(sc){}
                void operator()(){
```

```
                        std::unique_lock<std::mutex>(sc->objectListMutex);
                        std::list<Task*> tasks;
                        std::for_each(sc->objects.begin(), sc->objects.end(),
                                [&tasks](GameObject* go){
                                tasks.push_back(go->render());
                        });
                        TaskManager::pushTaskList(tasks);
                }
        private:
                /** Se pelialue, jonka oliot piirretään*/
                Scene* sc;
        };
};
}
#endif
```

SceneManager.h:

```
#ifndef SCENEMANAGER_H
#define SCENEMANAGER_H

#include "stdafx.h"
#include <vector>
#include "Scene.h"
#include <mutex>
#include <algorithm>
#include <iostream>
#include <string>

namespace stage_11{
        /** Pelialueita hallinnoiva olio*/
        class SceneManager{
        public:
                /** Luo uuden pelialueita hallinnoivan olion*/
                SceneManager(){}
                /** Tuhoaa pelialueita hallinnoivan olion ja sen sisältämät
                pelialueet*/
                ~SceneManager(){
                        std::unique_lock<std::mutex> lock(sceneListMutex);
                        std::for_each(scenes.begin(), scenes.end(), [](Scene* sc)
                                {delete sc; });
                }
                /** Luo uuden pelialueen
                @returns        Viite luotuun pelialueeseen
                */
                Scene& createScene(){
                        std::unique_lock<std::mutex> lock(sceneListMutex);
                        Scene* sc = new Scene();
                        scenes.push_back(sc);
                        return *sc;
                }
                /** Asettaa aktiivisen pelialueen, eli määrittelee minkä pelialueen
                tila päivitetään pelisilmukkaa suoritettaessa
                @param sc       Uuden aktiivisen pelialueen indeksi pelialuelistassa
                */
                void setActiveScene(unsigned int sc){
                        std::unique_lock<std::mutex> lock(sceneListMutex);
                        if (sc < 0 || sc > scenes.size()){
                                std::cerr << "Attempted to set invalid active scene;
                                        " << std::to_string(sc) << std::endl;
                                abort();
```

```
                }
                activeScene = scenes[sc];
        }
    protected:
        /** Lista pelin pelialueista*/
        std::vector<Scene*> scenes;
        /** Nykyinen aktiivinen pelialue*/
        Scene* activeScene;
        /** Pelialuelistaa suojaava lukko*/
        std::mutex sceneListMutex;
    };
}

#endif
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

Transform.h:

```
#ifndef TRANSFORM_H
#define TRANSFORM_H

#include "stdafx.h"
#include "Component.h"
#include <glm\glm.hpp>
#include <glm\gtc\matrix_transform.hpp>
#include <mutex>

#define TRANSFORM_ID 1

/** Peliolion sijaintia 3D-pelimaailmassa ylläpitävä komponentti*/
namespace stage_11{
    class Transform : public Component{
    public:
        /** Luo uuden sijaintikomponentin.
        @param owner  Komponentin omistava peliolio
        @param matrix Peliolion sijainti simulaation alussa
        */
        Transform(GameObject& owner, const glm::mat4& matrix = glm::mat4())
            : Component(owner), transform(matrix){}
        /** Hakee peliolion sijaintimatriisin
        @returns      Peliolion sijaintia ja orientaatiota esittävä 4x4-
                      matriisi
        */
        glm::mat4 getMatrix(){
            std::unique_lock<std::mutex>(tfmutex);
            return transform;
        }
        /** Asettaa peliolion sijaintimatriisin
        @param matrix Peliolion sijaintia ja orientaatiota esittävä 4x4-
                      matriisi
        */
        void setMatrix(const glm::mat4& matrix){
            std::unique_lock<std::mutex>(tfmutex);
            transform = matrix;
```

```
		}
		/** Hakee peliolion sijaintivektorin
		@returns	Peliolion sijaintia esittävä kolmipaikkainen vektori
		*/
		glm::vec3 getPosition(){
			std::unique_lock<std::mutex>(tfmutex);
			return glm::vec3(transform[3]);
		}
		/** Siirtää pelioliota haluttuun suuntaan
		@param direction	Vektori, joka ilmoittaa mihin suuntaan ja
					miten kauas pelioliota siirretään
		*/
		void translate(glm::vec3 direction){
			std::unique_lock<std::mutex>(tfmutex);
			transform = glm::translate(transform, direction);
		}
		/** Hakee sijaintikomponentin komponenttitunnuksen
		@returns	Sijaintikomponentin komponenttitunnus
		*/
		virtual int id(){
			return TRANSFORM_ID;
		}
	private:
		/** Peliolion sijaintia ja orientaatiota kuvaava 4x4-matriisi*/
		glm::mat4 transform;
		/** Sijaintimatriisia suojaava lukko*/
		std::mutex tfmutex;
	};
}
#endif
```

Moduuli Stage_11_thread:

LogManager.h:

```
#ifndef LOGMANAGER_H
#define LOGMANAGER_H

#include "stdafx.h"
#include <mutex>
#include <Logger.h>

namespace stage_11{
	/** Olio, joka hallinnoi säieturvallista kirjoitusta lokiin*/
	class LogManager{
	public:
		/** Luo uuden lokinhallintaolion
		@param standard	Kanava, johon kirjoitetaan tavalliset
					lokiviestit
		@param error	Kanava, johon kirjoitetaan virheilmoitukset
		*/
		LogManager(std::ostream& standard, std::ostream& error) :
			logger(standard, error){
		}
		/** Kirjoittaa lokiin tavallisen viestin
		@param msg	Kirjoitettava viesti
		*/
		void log(std::string msg){
			std::unique_lock<std::mutex> lock(logmutex);
			logger.Log(msg);
		}
```

```
		/** Kirjoittaa lokiin virheilmoituksen
		@param msg	Kirjoitettava viesti
		*/
		void logError(std::string msg){
			std::unique_lock<std::mutex> lock(logmutex);
			logger.LogError(msg);
		}
	private:
		LogManager(const LogManager& other) = delete;
		LogManager* operator= (const LogManager& other) = delete;

		/** Pääsyä lokiin hallinnoiva lukko*/
		std::mutex logmutex;
		/** Lokiolio*/
		stage_common::Logger logger;
	};
}
#endif
```

stdafx.h:

```
#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN
```

Task.h:

```
#ifndef TASK_H
#define TASK_H

#include "stdafx.h"

#define TASK_EXECUTE virtual void operator() ()

namespace stage_11{
	/**Olio, joka määrittelee säiealtaan työtehtävän.
	Pelimoottorin toimiessa säiealtaan työntekijäsäikeet hakevat työtehtäviä
	yksi kerrallaan ja suorittavat ne.*/
	class Task{
	public:
		/** Metodi, joka suorittaa työtehtävän laskennan*/
		TASK_EXECUTE = 0;
	};
}
#endif
```

TaskManager.h:

```
#ifndef TASKMANAGER_H
#define TASKMANAGER_H

#include "stdafx.h"
#include <iostream>
#include "TaskPool.h"
#include "LogManager.h"

/** Makro, jolla kirjoitetaan lokiin tavallinen lokiviesti*/
#define LOGMSG(MSG) stage_11::TaskManager::getGlobalLogger().log(MSG);
/** Makro, jolla kirjoitetaan lokiin virheviesti*/
#define LOGERR(MSG) stage_11::TaskManager::getGlobalLogger().logError(MSG);
```

```
namespace stage_11{
	/** Globaali singleton-olio, joka mahdollistaa työtehtävien lisäämisen
	säiealtaan suoritettavaksi.*/
	class TaskManager{
	public:
		/** Luo uuden työtehtävien hallintaolion
		@param threadcount   Säiealtaan työntekijäsäikeiden määrä
		*/
		TaskManager(unsigned int threadcount);
		/** Tuhoaa työntekijöiden hallintaolion*/
		~TaskManager();
		/** Lisää uuden työtehtävän säiealtaan suoritettavaksi
		@param t	Suoritettava työtehtävä
		*/
		static void pushTask(Task* t){
			singleton->tp.pushTask(t);
		}
		/** Lisää useita uusia työtehtäviä säiealtaan suoritettavaksi
		@param t	Suoritettavat työtehtävät
		*/
		static void pushTaskList(std::list<Task*>& t){
			singleton->tp.pushTaskList(t);
		}
		/** Hakee viitteen globaaliin työtehtävienhallintaolioon
		@returns	Viite globaaliin TaskManager-singletoniin
		*/
		static TaskManager& getSingleton(){
			return *singleton;
		}
		/** Hakee viitteen oletuslokiolioon
		@returns	Viite globaaliin lokiolioon
		*/
		static LogManager& getGlobalLogger(){
			return singleton->globalLogger;
		}
	private:
		/** Globaali TaskManager-singleton*/
		static TaskManager* singleton;
	protected:
		/** Säieallas*/
		TaskPool tp;
		/** Globaali lokiolio*/
		LogManager globalLogger;
	};
}
#endif
```

TaskManager.cpp:

```
#include "stdafx.h"
#include "TaskManager.h"

using namespace stage_11;

TaskManager* TaskManager::singleton = nullptr;

TaskManager::TaskManager(unsigned int threadcount) : tp(threadcount),
	globalLogger(std::cout, std::cerr){
	//Sallitaan vain yksi globaali TaskManager
	if (singleton != nullptr){
		globalLogger.logError(std::string("Error: attempted to create a
```

```
                        global Task Manager when one already exists"));
                abort();
        }
        singleton = this;
}
TaskManager::~TaskManager(){
        singleton = nullptr;
}
```

TaskPool.h:

```
#ifndef TASKPOOL_H
#define TASKPOOL_H

#include "stdafx.h"
#include "Task.h"
#include <list>
#include <mutex>
#include <condition_variable>
#include <algorithm>

namespace stage_11{
        /** Pelimoottorin pohjana toimiva säieallas*/
        class TaskPool{
        public:
                TaskPool& operator=(const TaskPool& other) = delete;
                TaskPool(const TaskPool& other) = delete;

                /** Hakee seuraavaksi suoritettavan työtehtävän.
                Jos suoritettavia työtehtäviä ei ole, jää odottamaan niiden
                saapumista.
                @returns        Osoitin suorittamattomaan työtehtävään. Tuhottava
                                suorittamisen jälkeen deletellä.
                */
                Task* pullTask();
                /** Lisää uuden työtehtävän suoritettavaksi.
                @param task     Osoitin suoritettavaan työtehtävään tai nullptr, jos
                                säieallas on pysäytetty
                */
                void pushTask(Task* task);
                /** Lisää uusia työtehtäviä suoritettavaksi.
                @param task     Lista, joka sisältää osoittimia suoritettaviin
                                työtehtäviin
                */
                void pushTaskList(std::list<Task*>& taskList);
                /** Odottaa, kunnes kaikki työtehtävät on suoritettu*/
                void waitForAllDone();
                /** Suorittaa jatkuvasti työtehtäviä, kunnes säieallas
                pysäytetään*/
                void work();
                /** Pysäyttää säiealtaan*/
                void terminate();
                /** Luo uuden säiealtaan
                @param threadCount      Työntekijäsäikeiden määrä
                */
                TaskPool(unsigned int threadCount) : threadCount(threadCount){}
                /** Tuhoaa säiealtaan
                Kutsu aina terminate()-metodia ja odota säikeiden suorituksen
                loppumista ennen tuhoamista*/
                ~TaskPool(){
                        //Tuhotaan suorittamattomat työtehtävät
```

```
                        std::for_each(tasks.begin(), tasks.end(), [](Task* t){delete
                                t; });
                }
        private:
                /** Lista suorittamattomista työtehtävistä*/
                std::list<Task*> tasks;
                /** Ilmoittaa, että työtehtäviä on saatavillla*/
                std::condition_variable hasTasks;
                /** Ilmoittaa, että kaikki työtehtävät on suoritettu*/
                std::condition_variable allDone;
                /** Työtehtävälistaa suojaava lukko*/
                std::mutex taskListMutex;

                /** Onko säiealtaan suoritus pysäytetty*/
                bool terminated = false;
                /** Niiden työntekijäsäikeiden määrä, joilla ei ole mitään
                tekemistä*/
                unsigned int waitingThreads = 0;
                /** Työntekijäsäikeiden kokonaismäärä*/
                unsigned int threadCount;
        };
}
#endif
```

TaskPool.cpp:

```
#include "stdafx.h"
#include "TaskPool.h"

using namespace stage_11;

Task* TaskPool::pullTask(){
        std::unique_lock<std::mutex> lock(taskListMutex);
        //Jos työtehtäviä ei ole, jäädään odottamaan
        if (tasks.size() < 1 && !terminated){
                waitingThreads++;
                //Jos kaikki säikeet odottavat, herätetään pelisilmukka
                if (waitingThreads >= threadCount) allDone.notify_one();
                hasTasks.wait(lock, [this]{return tasks.size() > 0 ||
                        terminated; });
                waitingThreads--;
        }
        //Säieallas pysäytetty, lopetetaan suoritus
        if (terminated) return nullptr;
        //Palautetaan ensimmäinen työtehtävä
        Task* ret = tasks.front();
        tasks.pop_front();
        return ret;
}
void TaskPool::pushTask(Task* task){
        std::unique_lock<std::mutex> lock(taskListMutex);
        tasks.push_back(task);
        lock.unlock();
        //Herätetään työtehtäviä odottava säie
        hasTasks.notify_one();
}
void TaskPool::pushTaskList(std::list<Task*>& taskList){
        std::unique_lock<std::mutex> lock(taskListMutex);
        for (std::list<Task*>::iterator it = taskList.begin(); it !=
                taskList.end(); it++){
                tasks.push_back(*it);
```

```
        }
        lock.unlock();
        //Herätetään työtehtäviä odottavat säikeet
        hasTasks.notify_all();
}
void TaskPool::waitForAllDone(){
        std::unique_lock<std::mutex> lock(taskListMutex);
        //Odotetaan, kunnes kaikki säikeet ovat odotustilassa ja tehtävälista on
        //tyhjä
        if (waitingThreads < threadCount || tasks.size() > 0) allDone.wait(lock,
                [this]{
                return (waitingThreads == threadCount && tasks.size() < 1);
        });
}
void TaskPool::work(){
        while (true){
                //Haetaan uusi työtehtävä
                Task* task = pullTask();
                //Lopetetaan suoritus, kun tehtäviä ei ole
                if (task == nullptr) return;
                //Suoritetaan haettu tehtävä
                (*task)();
                //Tuhotaan suoritettu tehtävä
                delete task;
        }
}
void TaskPool::terminate(){
        std::unique_lock<std::mutex> lock(taskListMutex);
        terminated = true;
        lock.unlock();
        //Herätetään työntekijäsäikeet, jotta niiden suoritus voidaan lopettaa
        hasTasks.notify_all();
}
```